Supplementary Material for: Homogeneity and identity tests for unidimensional Poisson processes with an application to neurophysiological peri-stimulus time histograms–Python version.

Christophe Pouzat[1],
Antoine Chaffiol[2],
Avner Bar-Hen[1]
[1] MAP5, Paris-Descartes University and CNRS UMR 8145
[2] Pierre and Marie Curie University, CNRS UMR 7210 and INSERM UMR U968

November 4, 2015

Contents

1 Introduction

This document presents the analysis with `Python`. The exposition follows roughly the software development approach used in this project. Namely, a single PSTH is analyzed first step by step, requiring the definitions of short functions or the use of a few command lines. Once this prototypical analysis is achieved, one class and its associated methods are defined. The code of the methods being the same (modulo some variable name changes) as the code of the functions previously defined. For clarity of the code presentation—as well as to keep the code length able to fit within a single page—the literate programming paradigm is used throughout this document, implying that the construction of the actual working code often implies sticking together several pieces. Therefore many listings, like Listing 1, will appear like:

```
Some code lines in R or Python
<<a-reference>>
Some more code lines
```

In such cases a "reference" made of a string between "«" and "»" (in the case above "a-reference") refers to a listing whose content should be copied and pasted in place of the reference.

Figures, tables and equations numbers given in this document refer to figures, tables and equations in the companion manuscript.

1.1 Existing tests

Cox and P. A. W. Lewis (1966) present tests for homogeneous Poisson (Sec. 6.3) and renewal (Sec. 6.4) processes. The tests for Poisson processes use the fact that if the observed times: $\{t_1, t_2, \ldots, t_n\}$ are a realization of a homogeneous Poisson process with rate λ on the time interval $[0, t_0]$, then,

conditionally on n, the total number of events observed at the end of the time period, the quantities: $\{u_{(i)} = t_i/t_0\}_{i=1,\ldots,n}$ are observations of the order statistics of n IID draws from a uniform distribution on $(0,1)$. It is then possible to apply a Kolmogorov test or an Anderson-Darling test against this null hypothesis giving a *uniform conditional test for a Poisson process*. Durbin (1961, p. 48) followed by Peter A. W. Lewis (1965) argue further for the use of what Cox and P. A. W. Lewis (1966, p. 154-155) dubbed *Durbin's transformation* of the t_i in order to improve the power of these tests against the uniform null hypothesis. The algorithm producing this transformation follows:

1. Go from the $\{u_{(i)} = t_i/t_0\}_{i=1,\ldots,n}$ discussed in the previous paragraph to the intervals: $c_1 = u_{(1)}$, $c_i = u_{(i)} - u_{(i-1)}$ $(i = 2,\ldots,n)$, $c_{n+1} = 1 - u_{(n)}\}$ (the latter should IID realizations from an exponential distribution with parameter 1).

2. Get the order statistics $\{c_{(1)},\ldots,c_{(n)}\}$ and form the differences $g_i = (n+2-i)\left(c_{(i)} - c_{(i-1)}\right)$ for $i = 1,\ldots,n+1$ with $c_{(0)} = 0$ (they should be independent exponentially distributed random variables with means 1).

3. The observations $u'_{(i)} = \sum_{j=1}^{i} g_j$ for $i = 1,\ldots,n$ should then be observations from the order statistics of n IID draws from a uniform distribution on $(0,1)$.

As pointed out by Cox and P. A. W. Lewis (1966, p. 158) the tests on transformed data are sensitive to discretization: they fail to apply if the latter is too coarse. The data used here where sampled at 12800 Hz with a spike sorting procedure that did not properly cope with sampling jitter (Pouzat and Detorakis 2014). This unaccounted for sampling jitter amounts to a "too coarse" sampling and give rise to a pronounced stair-case aspect of the empirical cumulative distribution function (ECDF) of the $u'_{(i)}$ for small values of i. This leads to spurious positive values when applying the Anderson-Darling test. We therefore decided when working with the transformed data to jitter the original observed times uniformly by plus or minus half a sampling period (in practice plus or minus 40 μs). This destroys the stair-case aspect without touching the overall structure.

In addition to these tests against a uniform distribution on $(0,1)$, the correlation coefficients of the successive inter-event intervals at different lags (the autocorrelation function of the inter-events intervals) is inspected and the log of the survivors function—that should be a straight line under the null hypothesis—is plotted.

1.2 A remark on the pseudo-random number generators used by `R` and `Python`

As most readers know, when a (pseudo) random number is drawn from a continuous distribution (exponential, normal, etc) a function of one or several random numbers with a uniform distribution on [0,1) is used: exponential random numbers are typically generated with the inversion method–this is done in both `R` with `rexp` and in the `numpy.random` module of `Python` with `exponential`–; normal random numbers are generated with the inversion method–used by default in function `rnorm` of `R`–or with the Box-Muller method–used by function `normal` in `numpy.random`–or with the Kinderman and Monahan method, etc. This implies that a crucial role is played by the generator of uniform random numbers on [0,1)–or (0,1) as is the case for `R`–. In principle, when one reads the documentation of the *default* uniform pseudo-random number generators (`PRNG`) implemented in both `R` and `Python`, one gets the impression they are the same since both software used the Mersenne Twister. This PRNG generates in fact discrete number in $\{0, 1, \ldots, 2^{32} - 1\}$ with a period of $2^{19937} - 1$. This feat is achieved by using a tuple with 624 elements, each element being an unsigned integer coded on 32 bit. This means that such a tuple has to be provided in order to initialize the generator. `R` and `Python` do this initialization differently and in order to figure out precisely how they do it, the source codes have to be inspected. It is then possible (but tedious) to use the same tuple in both languages. Then one realizes that the generated sequences of *floating point numbers* (uniform on the unit interval) are different! Inspection of the source codes provides again the explanation: at each call, the Mersenne Twister outputs an unsigned integer coded on 32 bit; `R` divides this number by 2^{32} to get a floating point number $\in [0, 1)$ –then `R` checks if the number is 0 (or negative) and in such a case it returns $1/2 \times 1/(2^{32} - 1)$ –; `Python` draws *two successive* numbers from the Mersenne-Twister and constructs an "intermediate" 53 bit unsigned integer with them–the leftmost 27 bit of first 32 bit unsigned integer provide the leftmost 27 bit of the intermediate number while the leftmost 26 bit of the second 32 bit unsigned integer provide the rightmost 26 bit of the intermediate number; the intermediate number is then divided by 2^{53} to yield a floating point number $\in [0, 1)$ (with the maximal achievable resolution with double precision). `R` generates therefore double precision floating point random numbers with a 32 bit resolution, while `Python` generates numbers with a 53 bit resolution. This (undocumented) difference does not create significant differences in the two versions of our code but it explains why we could not work with the exact same sequences in both versions.

2 The analysis with Python

2.1 Setting up Python

The analysis presented in the manuscript and detailed next is carried out with `Python 3` (the following code runs and gives identical results with `Python 2`). We are going to use the 3 classical modules of Python's scientific ecosystem: `numpy`, `scipy` and `matplotlib`. We are also going to use two additional modules: `sympy` as well as `h5py`. We start by importing these modules:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy
import sympy as sy
import h5py
```

2.2 Implementation of existing tests

We define a function returning the Kolmogorov two sided or one sided statistics against the null hypothesis—uniform distribution on $(0, 1)$:

```
def Kolmogorov_D(Up,
                 what="D"):
    import numpy as np
    if not np.all(Up > 0) and np.all(Up < 1):
        raise ValueError('Every u in Up must satisfy 0 < u < 1')
    if not what in ["D","D+","D-"]:
        raise ValueError('what must be one of "D","D+","D-"')
    n = len(Up)
    ecdf = np.arange(1,n+1)/n
    Up = np.sort(Up)
    Dp = np.max(ecdf-Up)*np.sqrt(n)
    Dm = np.max(Up[:-1]-ecdf[:-1]+1/n)*np.sqrt(n)
    if what == "D": return max(Dp,Dm)
    if what == "D+": return Dp
    if what == "D-": return Dm
```

```
def pDsN(z,k_max=9):
  ## Returns the asymptotic value of CDF of the Kolmogorov
  ## $D_n\sqrt(n)$ statistics: the maximal distance between
  ## the theoretical CDF and the empirical one multiplied
  ## by the square root of the sample size.
  ## The asymptotic formula of Kolmogorov is used
  ## Euler's acceleration with van Wijngaarden transformaion is used
  import numpy as np
  partial_sum = np.cumsum((-1)**np.arange(1,k_max+1)*\
                          np.exp(-2*np.arange(1,k_max+1)**2*z**2))
  for i in range(1,int(round(k_max*2/3))):
    partial_sum = (partial_sum[1:]+partial_sum[:-1])/2
  return 1+2*partial_sum[-1]
```

We define next a function returning the Anderson-Darling statistics against the same null hypothesis:

```
def AndersonDarling_W2(Up):
    import numpy as np
    if not np.all(Up > 0) and np.all(Up < 1):
        raise ValueError('Every u in Up must satisfy 0 < u < 1')
    n = len(Up)
    return -n-np.sum((2*np.arange(1,n+1)-1)*np.log(Up)+\
                (2*n-2*np.arange(1,n+1)+1)*np.log(1-Up))/n
```

There are few published tables of the cumulative distribution function of the Anderson-Darling statistics (either for finite sample size or in the asymptotic limit) and there is no R function returning it. The G. Marsaglia and J. Marsaglia (2004, page 3) algorithm returning this function with sixth decimal place (or more) precision is therefore implemented next:

```
def pAD_W2(x):
  ## Marsaglia and Marsaglia (2004) JSS 9(2):1--5
  if x<=0: return 0
  if 0 < x < 2:
    res = 1/np.sqrt(x)*np.exp(-1.2337141/x)
    res *= (2.00012+\
            (.247105-\
             (.0649821-\
              (.0347962-\
               (.011672-.00168691*x)*x)*x)*x)*x)
    return res
  res = 1.0776-(2.30695-\
                (.43424-\
                 (.082433-\
                  (.008056-.0003146*x)*x)*x)*x)*x
  res = np.exp(-np.exp(res))
  return res
```

We can test this implementation using the 0.9, 0.95 and 0.99 quantiles given by G. Marsaglia and J. Marsaglia (2004, page 2):

```
print(" Nominal 0.90, computed: ",pAD_W2(1.9329578327),", difference: ",
      0.9-pAD_W2(1.9329578327),"\n",
      "nominal 0.95, computed: ",pAD_W2(2.492367),", difference: ",
      0.95-pAD_W2(2.492367),"\n",
      "nominal 0.99, computed: ",pAD_W2(3.878125),", difference: ",
      0.99-pAD_W2(3.878125))
```

```
Nominal 0.90, computed:  0.899988917447 , difference:  1.10825528162e-05
nominal 0.95, computed:  0.950008128363 , difference:  -8.12836257569e-06
nominal 0.99, computed: 0.989997384292 , difference:  2.61570839077e-06
```

The function performing Durbin's transformation is defined next. It takes a series of observed times and an observation interval as arguments:

```
def DurbinTransform(observed_times,
                    observation_interval=None):
    import numpy as np
    if observation_interval == None:
        observation_interval = [np.floor(np.min(observed_times)),
                                np.ceil(np.max(observed_times))]
    if not np.all(np.logical_and(observation_interval[0] < observed_times,
                                 observed_times < observation_interval[1])):
        raise ValueError('observation_interval is not compatible with'+\
                         'observed_times')
    observed_times = observed_times.copy()-observation_interval[0]
    obs_duration = np.diff(observation_interval)
    n = len(observed_times)
    observed_times /= obs_duration
    iei = [observed_times[0]]+\
      list(np.sort(np.diff(observed_times)))+\
      [1-observed_times[-1]]
    siei = [0]+sorted(iei)
    g = (n+2-np.arange(1,n+2))*np.diff(siei)
    return np.cumsum(g[:-1])
```

2.2.1 An exploration of time discretization and jittering effects on these statistics

As discussed in the first section, the time discretization due to sampling at acquisition time and sub-optimal spike sorting algorithm has consequences on the statistics used to test if an observed (aggregated process) is homogeneous Poisson or not. These consequences are explored here with simulations mimicking the pre-stimulation period of neuron 2 from data set `e070528citronellal` whose analysis is presented in the sequel. This neurons fires 1455 during 6 seconds (and 15 trials) giving an aggregated rate of 242.5 Hz. We perform next a simulation of 10000 homogeneous Poisson processes

with the latter rate during 6 s. The two sided Kolmogorov statistics–$D_n\sqrt{n}$ whose 0.95 and 0.99 quantiles are 1.358 and 1.628 respectively–as well as the Anderson-Darling one–W_n^2 whose 0.95 and 0.99 quantiles are 2.492 and 3.878 respectively, the correct value of the latter quantile is from G. Marsaglia and J. Marsaglia (2004, page 2)–are computed on the resulting conditionally uniform process ($\{t_1/6, \ldots, t_n/6\}$) as well as on its discretized version (with a time resolution corresponding to the actual sampling period of our data sets, 1/12800 s) and on a time jittered version of the discretized version (with a uniform jitter between -1/2 and +1/2 the sampling period). The same is done on the data after Durbin's transformation. A function is defined first doing the discretization:

```
def discretize_time(observed_times,
                    observation_period=[0,6],
                    sampling_period=1/12800):
    import numpy as np
    dt = np.arange(observation_period[0],
                   observation_period[1],
                   sampling_period)
    return (0.5+(np.digitize(observed_times,dt)-1))*sampling_period
```

A function doing the time jittering is defined next (taking care of the events sitting close to the observation interval boundaries):

```
def jitter_time(observed_times,
                observation_period=[0,6],
                sampling_period=1/12800):
    import numpy as np
    from numpy.random import random_sample
    n = len(observed_times)
    res = np.zeros(n)
    within = np.logical_and(observation_period[0]+\
                            sampling_period/2 < observed_times,
                            observed_times < observation_period[1]-\
                            sampling_period/2)
    res[within] = observed_times[within]+\
      (random_sample(sum(within))-0.5)*sampling_period
    too_small = observation_period[0]+sampling_period/2 >= observed_times
    if sum(too_small) > 0:
        s = random_sample(sum(too_small))
        s *= (observed_times[too_small]+sampling_period/2-\
              observation_period[0]*np.ones(len(s)))
        s += observation_period[0]*np.ones(len(s))
        res[too_small] = s
    too_big = observed_times >= observation_period[1]-sampling_period/2
    if sum(too_big)>0:
        b = random_sample(sum(too_big))
        b *= (observation_period[1]*np.ones(len(b))-\
              observed_times[too_big]-sampling_period/2)
        b += observed_times[too_big]-sampling_period/2
        res[too_big] = b
    res[res==0] = 5*np.finfo(float).eps
    return np.sort(res)
```

We can now do the simulation with a single realization as follows:

```
from numpy.random import seed, exponential
seed(20110928)
hp1 = np.cumsum(exponential(1/242.5,2000))
hp1 = hp1[hp1<6]
hp1_d = discretize_time(hp1)
hp1_dj = jitter_time(hp1_d)
```

The Kolmogorov and Anderson-Darling statistics are:

```
D_W2_1 = {"D_o":Kolmogorov_D(hp1/6),
          "D_d":Kolmogorov_D(hp1_d/6),
          "D_dj":Kolmogorov_D(hp1_dj/6),
          "W2_o":AndersonDarling_W2(hp1/6),
          "W2_d":AndersonDarling_W2(hp1_d/6),
          "W2_dj":AndersonDarling_W2(hp1_dj/6)}
res_out = "\n       original   discretized      jittered\n"
res_out += "D  {D_o:12.8f}  {D_d:12.8f}  {D_dj:12.8f}\n"
res_out += "W2 {W2_o:12.8f}  {W2_d:12.8f}  {W2_dj:12.8f}"
print(res_out.format(**D_W2_1))
```

```
       original   discretized      jittered
D    1.11505243    1.11485942    1.11483285
W2   1.28243421    1.28239418    1.28238885
```

There is no "huge" effect of time discretization here. The same is done after Durbin's transformation. Since intervals of length 0 can be obtained with the discretized data, we set these zero length intervals to five times the smallest floating point number the machine can represent:

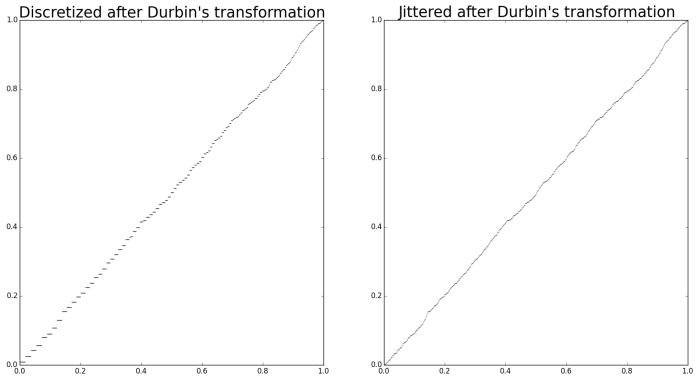
```
hp1_dt = DurbinTransform(hp1,[0,6])
hp1_d_dt = DurbinTransform(hp1_d,[0,6])
if np.any(hp1_d_dt==0):
    hp1_d_dt[hp1_d_dt==0] = 5*np.finfo(float).eps

if np.any(hp1_d_dt==1):
    hp1_d_dt[hp1_d_dt==1] -= 5*np.finfo(float).eps

hp1_dj_dt = DurbinTransform(hp1_dj,[0,6])
```

The Kolmogorov and Anderson-Darling statistics are then:

```
D_W2_2 = {"D_o":Kolmogorov_D(hp1_dt),
          "D_d":Kolmogorov_D(hp1_d_dt),
          "D_dj":Kolmogorov_D(hp1_dj_dt),
          "W2_o":AndersonDarling_W2(hp1_dt),
          "W2_d":AndersonDarling_W2(hp1_d_dt),
          "W2_dj":AndersonDarling_W2(hp1_dj_dt)}
print(res_out.format(**D_W2_2))
```

```
       original   discretized      jittered
D    0.57403286    0.76730289    0.56164707
W2   0.47769627    3.96781058    0.47785563
```

There is a large effect of discretization on Anderson-Darling's statistics; effect that seems to be canceled by adding a jitter. Making a figure with the corresponding empirical cumulative distribution functions can help here; the stair-case pattern is very clear on the ECDF of the discretized data after Durbin's transformation:



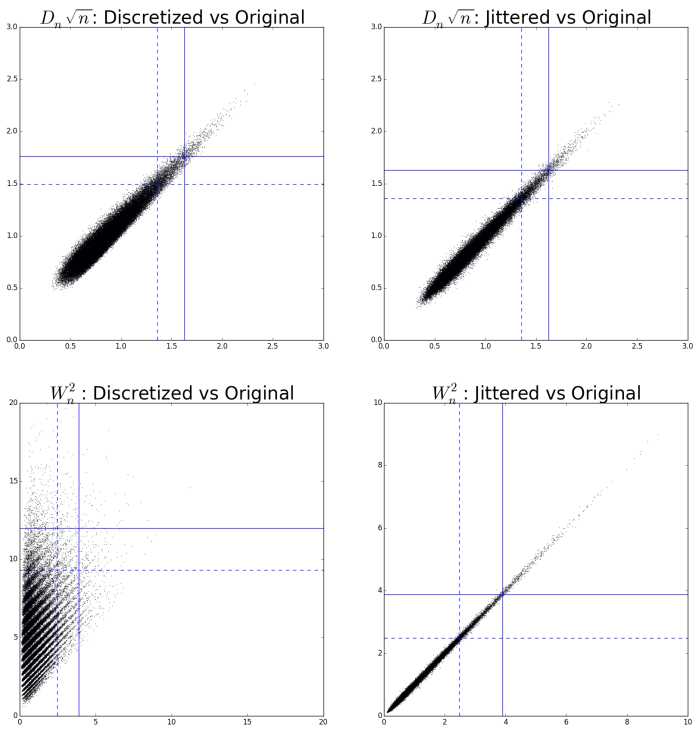
A systematic simulation is done as follows–printing the empirical 0.95 and 0.99 quantiles for each statistic at the end–:

```
seed(20110928)
nrep = 50000
D_W2 = np.zeros((nrep,12))
for i in range(nrep):
  hp = np.cumsum(exponential(1/242.5,2000))
  hp = hp[hp<6]
  hp_d = discretize_time(hp)
  hp_dj = jitter_time(hp_d)
  hp_dt = DurbinTransform(hp,[0,6])
  hp_d_dt = DurbinTransform(hp_d,[0,6])
  if np.any(hp_d_dt<=0):
    hp_d_dt[hp_d_dt<=0] = 5*np.finfo(float).eps
  if np.any(hp_d_dt>=1):
    hp_d_dt[hp_d_dt>=1] = 1-5*np.finfo(float).eps
  hp_dj_dt = DurbinTransform(hp_dj,[0,6])
  if np.any(hp_dj_dt<=0):
    hp_dj_dt[hp_dj_dt<=0] = 5*np.finfo(float).eps
  if np.any(hp_dj_dt>=1):
    hp_dj_dt[hp_dj_dt>=1] = 1-5*np.finfo(float).eps
  D_W2[i,:] = [Kolmogorov_D(hp/6),Kolmogorov_D(hp_d/6),
               Kolmogorov_D(hp_dj/6),AndersonDarling_W2(hp/6),
               AndersonDarling_W2(hp_d/6),AndersonDarling_W2(hp_dj/6),
               Kolmogorov_D(hp_dt),Kolmogorov_D(hp_d_dt),
               Kolmogorov_D(hp_dj_dt),AndersonDarling_W2(hp_dt),
               AndersonDarling_W2(hp_d_dt),AndersonDarling_W2(hp_dj_dt)]
```

```
Data without Durbin's transformation
================================================
  Empirical quantile at:        0.95      0.99
  ------------------------------------------------
  Raw data D:                   1.3530    1.6091
  Discretized data D:           1.3530    1.6091
```

```
  Jittered data D:              1.3531    1.6090
  Raw data W2:                  2.4906    3.8500
  Discretized data W2:          2.4905    3.8500
  Jittered data W2:             2.4905    3.8500

Data with Durbin's transformation
================================================

  Empirical quantile at:        0.95      0.99
  ----------------------------------------------
  Raw data D:                   1.3564    1.6265
  Discretized data D:           1.4930    1.7628
  Jittered data D:              1.3576    1.6291
  Raw data W2:                  2.4810    3.8897
  Discretized data W2:          9.3200    11.9809
  Jittered data W2:             2.4860    3.8776
```

Plotting the statistics for the discretized vs original and the "discretized and then jittered" vs original shows very clearly that the Anderson-Darling test *should not* be used for discretized data after Durbin's transformation but that jittering the discretized data makes the statistics behave essentially as the ones of the original data (the blue lines show the empirical 0.95 and 0.99 quantiles):



2.3 Getting the data

Our data (Pouzat and Chaffiol 2015) are stored in HDF5 format on the zenodo server (`DOI:10.5281/zenodo.1428145`). They are all contained in a file named `CockroachDataJNM_2009_181_119.h5`. The data within this file have an hierarchical organization similar to the one of a file system (one of the main ideas of the HDF5 format). The first organization level is the experiment; there are 4 experiments in the file: `e060517`, `e060817`, `e060824` and `e070528`. Each experiment is organized by neurons, `Neuron1`, `Neuron2`, etc, (with a number of recorded neurons depending on the experiment). Each neuron contains a `dataset` (in the HDF5 terminology) named `spont` containing the spike train of that neuron recorded during a period of spontaneous activity. Each neuron also contains one or several further sub-levels named after the odor used for stimulation `citronellal`, `terpineol`, `mixture`, etc. Each a these sub-levels contains as many `datasets`: `stim1`, `stim2`, etc, as

stimulations were applied; and each of these data sets contains the spike train of that neuron for the corresponding stimulation. Another `dataset`, named `stimOnset` containing the onset time of the stimulus (for each of the stimulations). All these times are measured in seconds.

The data can be downloaded with `Python` as follows:

```
try:
    from urllib.request import urlretrieve # Python 3
except ImportError:
    from urllib import urlretrieve # Python 2

name_on_disk = 'CockroachDataJNM_2009_181_119.h5'
urlretrieve('https://zenodo.org/record/14281/files/'+
            name_on_disk,
            name_on_disk)
```

2.4 Step by step analysis of the response of neuron 2 from data set `e070528citronellal`

2.4.1 Definition and tests of PSTH construction and stabilization

We define class `StabilizedPSTH` that contains a PSTH together with its variance stabilized version, after setting the stimulus onset time at 0. The skeleton of this class definition is:

```
class StabilizedPSTH:
    """Holds a Peri-Stimuls Time Histogram (PSTH) and
    its variance stabilized version.

    Attributes:
        st (1d array): aggregated spike trains (stimulus on at 0).
        x (1d array): bins' centers.
        y (1d array): stabilized counts.
        n (1d array): actual counts.
        n_stim (scalar): number of trials used to build
            the PSTH.
        width (scalar): bin width.
        stab_method (string): variance stabilization method.
        spontaneous_rate (scalar): spontaneous rate.
        support_length (scalar): length of the PSTH support.
    """
    <<init_StabilizedPSTH>>
    <<str_StabilizedPSTH>>
    <<plot_StabilizedPSTH>>
```

Listing 1: `StabilizedPSTH-class-definition-python`

`init_StabilizedPSTH` The constructor of StabilizedPSTH class instances is defined by:

```
def __init__(self,spike_train_list,
             onset,region = [-2,8],
             spontaneous_rate = None,target_mean = 3,
             stab_method = "Freeman-Tukey"):

    <<init_StabilizedPSTH_docstring>>
    import numpy as np
    if not isinstance(spike_train_list,list):
        raise TypeError('spike_train_list must be a list')
    n_stim = len(spike_train_list)
    aggregated = np.sort(np.concatenate(spike_train_list))
    if spontaneous_rate is None:
        time_span = np.ceil(aggregated[-1])-np.floor(aggregated[0])
        spontaneous_rate = len(aggregated)/n_stim/time_span
    if not spontaneous_rate > 0:
        raise ValueError('spontaneous_rate must be positive')
    if not stab_method in ["Freeman-Tukey","Anscombe","Brown et al"]:
        raise ValueError('stab_method should be one of '\
                         +'"Freeman-Tukey","Anscombe","Brown et al"')
    left = region[0]+onset
    right = region[1]+onset
    aggregated = aggregated[np.logical_and(left <= aggregated,
                                           aggregated <= right)]-onset
    bin_width = np.ceil(target_mean/n_stim/spontaneous_rate*1000)/1000
    aggregated_bin = np.arange(region[0],
                               region[1]+bin_width,
                               bin_width)
    aggregated_counts, aggregated_bin = np.histogram(aggregated,
                                                     aggregated_bin)
    if stab_method == "Freeman-Tukey":
        y = np.sqrt(aggregated_counts)+np.sqrt(aggregated_counts+1)
    elif stab_method == "Anscombe":
        y = 2*np.sqrt(aggregated_counts+0.375)
    else:
        y = 2*np.sqrt(aggregated_counts+0.25)
    self.st = aggregated
    self.x = aggregated_bin[:-1]+bin_width/2
    self.y = y
    self.n = aggregated_counts
    self.n_stim = n_stim
    self.width = bin_width
    self.stab_method = stab_method
    self.spontaneous_rate = spontaneous_rate
    self.support_length = np.diff(region)[0]
```

Listing 2: `init_StabilizedPSTH`

`init_StabilizedPSTH_docstring` It's *docstring* is:

```
""" Create a StabilizedPSTH instance.

Parameters
----------
spike_train_list: a list of spike trains (vectors with strictly
  increasing elements), where each element of the list is supposed
  to contain a response and where each list element is assumed
  time locked to a common reference time.
onset: a number giving to the onset time of the stimulus.
region: a two components list with the number of seconds before
  the onset (a negative number typically) and the number of second
  after the onset one wants to use for the analysis.
spontaneous_rate: a positive number with the spontaneous rate
  assumed measured separately; if None, the overall rate obtained
  from spike_train_list is used; the parameter is used to set the
  bin width automatically.
target_mean: a positive number, the desired mean number of events
  per bin under the assumption of homogeneity.
stab_method: a string, either "Freeman-Tukey" (the default,
  x -> sqrt(x)+sqrt(x+1)), "Anscombe" (x -> 2*sqrt(x+3/8)) or "Brown
  et al" (x -> 2*sqrt(x+1/4); the variance stabilizing transformation.
"""
```

Listing 3: `init_StabilizedPSTH_docstring`

`str_StabilizedPSTH` The string method (used by function `print`) for StabilizedPSTH:

```
def __str__(self):
    """Controls the printed version of the instance."""
    import numpy as np
    return "An instance of StabilizedPSTH built from " \
        + str(self.n_stim) + " trials with a " + str(self.width) \
        + " (s) bin width.\n  The PSTH is defined on a domain " \
        + str(self.support_length) + " s long.\n" \
        + "  The stimulus comes at second 0.\n" \
        + "  Variance was stabilized with the " \
        + self.stab_method + " method.\n"
```

Listing 4: `str_StabilizedPSTH`

`plot_StabilizedPSTH` We define now the plot method for StabilizedPSTH instances:

```
def plot(self,
         what="stab",
         linewidth=1,
         color='black',
         xlabel="Time (s)",
         ylabel=None):
        """Plot the data.

        Parameters
        ----------
        what: a string, either 'stab' (to plot the stabilized version)
              or 'counts' (to plot the actual counts).
        The other parameters (linewidth,color,xlabel,ylabel) have their
            classical meaning
        """
        import matplotlib.pyplot as plt
        if not what in ["stab","counts"]:
            raise ValueError('what should be either "stab" or "counts"')
        if what == "stab":
            y = self.y
            if ylabel is None:
                if self.stab_method == "Freeman-Tukey":
                    ylabel = r'$\sqrt{n}+\sqrt{n+1}$'
                elif self.stab_method == "Anscombe":
                    ylabel = r'$2 \sqrt{n+3/8}$'
                else:
                    ylabel = r'$2 \sqrt{n+1/4}$'
        else:
            y = self.n
            if ylabel is None:
                ylabel = "Counts per bin"
        plt.plot(self.x,y,color=color,linewidth=linewidth)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
```

Listing 5: `plot_StabilizedPSTH`

Tests We now test these functions and methods. We use the data recorded in the spontaneous to estimate the spontaneous discharge frequency:

```
f = h5py.File("CockroachDataJNM_2009_181_119.h5","r")
nu_spont_n2 = len(f["e070528/Neuron2/spont"])/60
print("The spontaneous rate of neuron 2 from experiment e070528 is: ",
      nu_spont_n2, "(Hz).")
```

```
The spontaneous rate of neuron 2 from experiment e070528 is:  19.55
```

We create the spike train list and extract the stimulus onset time:

```
citron_onset = f["e070528/Neuron2/citronellal/stimOnset"][...][0]
train_list = [f[y][...] for y in
              ["e070528/Neuron2/citronellal/stim"+str(x)
               for x in range(1,16)]]
```

We then build the instance of our new class `StabilizedPSTH` for neuron 2 of the data set; we also use the newly defined `print` method for this instance:

```
citron_spsth_n2 = StabilizedPSTH(train_list,
                                 spontaneous_rate=nu_spont_n2,
                                 region = [-6,6],
                                 onset=citron_onset)
print(citron_spsth_n2)
```

```
An instance of StabilizedPSTH built from 15 trials with a 0.011 (s) bin width.
  The PSTH is defined on a domain 12 s long.
  The stimulus comes at second 0.
  Variance was stabilized with the Freeman-Tukey method.
```

The `plot` method displaying the stabilized PSTH is invoked with:

```
citron_spsth_n2.plot()
```

The one displaying the actual counts is invoked with:

```
citron_spsth_n2.plot(what="counts")
```

Is the pre-stimulation period compatible with a homogeneous Poisson process? As mentioned in the companion manuscript the tests homogeneous / non-homogeneous Poisson we propose are valid only if the convergence to the Poisson process has been reached. This requires that responses to the successive stimulations were uncorrelated and that enough stimulations were aggregated to loose the "memory" exhibited by the individual responses (they are clearly not Poisson). As a first step we can check if the pre-stimulation period is compatible with the realization of a homogeneous Poisson process. We can perform what Cox and P. A. W. Lewis (1966) call a *uniform conditional test for a Poisson process* on the original data computing both the Kolmogorov and the Anderson-Darling statistics:

```
early_train = citron_spsth_n2.st[citron_spsth_n2.st < 0] + 6
et_stat = (Kolmogorov_D(early_train/6),
           AndersonDarling_W2(early_train/6))
print(("D: {D:.4}, Prob(D): {PD:.4f}\n"
       "W2: {W2:.4f}, Prob(W2): {PW2:.4f}").format(D=et_stat[0],
                                                   PD=pDsN(et_stat[0]),
                                                   W2=et_stat[1],
                                                   PW2=pAD_W2(et_stat[1])))
```

```
D: 0.8279, Prob(D): 0.5005
W2: 0.6256, Prob(W2): 0.3759
```

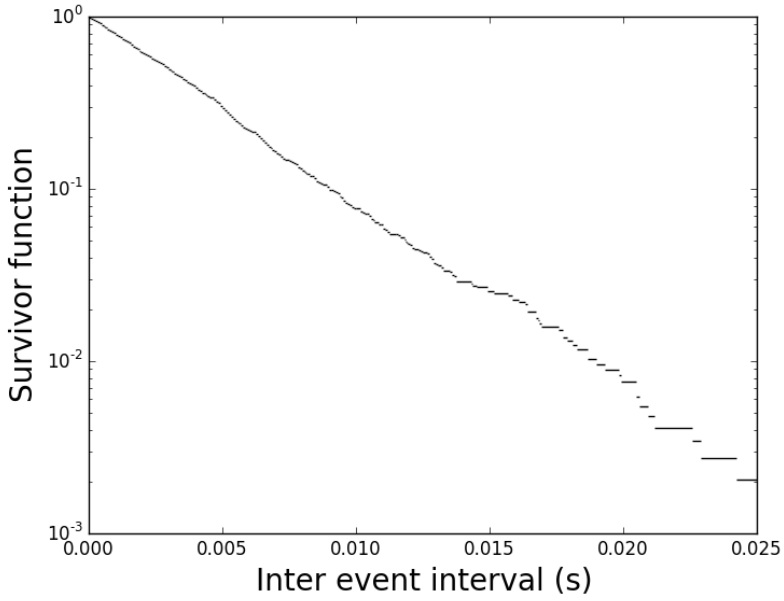
Working Durbin's transformation after jittering the data we get:

```
early_train_j = jitter_time(early_train,(0,6))
early_train_t = DurbinTransform(early_train_j,(0,6))
ett_stat = (Kolmogorov_D(early_train_t),
            AndersonDarling_W2(early_train_t))
print(("D: {D:.4}, Prob(D): {PD:.4f}\n"
       "W2: {W2:.4f}, Prob(W2): {PW2:.4f}").format(D=ett_stat[0],
                                                  PD=pDsN(ett_stat[0]),
                                                  W2=ett_stat[1],
                                                  PW2=pAD_W2(ett_stat[1])))
```

```
D: 1.09, Prob(D): 0.8140
W2: 2.0456, Prob(W2): 0.9133
```

We can obtain a plot of the log-survivor function of the intervals with:

```
iei_early = np.diff(early_train)
iei_early_s = np.sort(iei_early)
plt.hlines(y=1-(np.arange(len(iei_early))+1)/len(iei_early),
           xmin=[0]+list(iei_early_s[:-1]),
           xmax=iei_early_s)
plt.xlim(0,0.025)
plt.yscale('log')
plt.ylim(0.001,1)
plt.xlabel("Inter event interval (s)",fontdict={'fontsize':20})
plt.ylabel("Survivor function",fontdict={'fontsize':20})
plt.savefig('figs/e070528citronN2LogSurvPython.png')
plt.close()
'figs/e070528citronN2LogSurvPython.png'
```



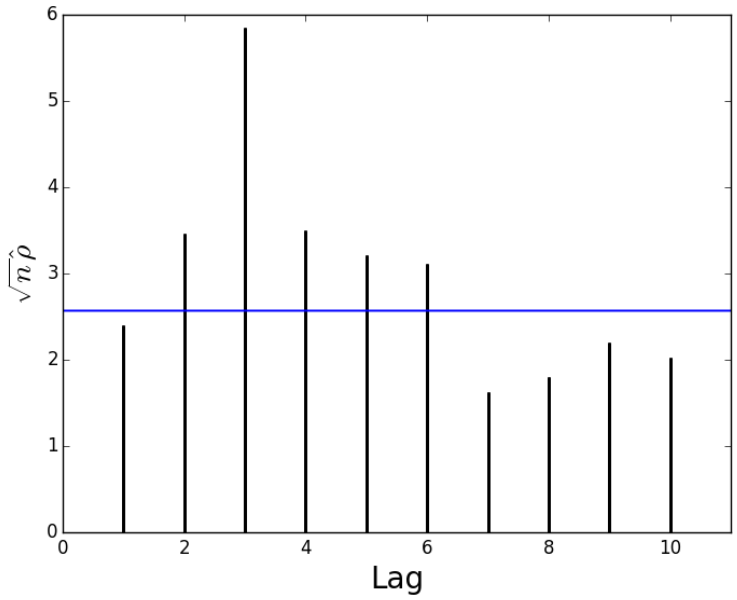
The auto-correlation coefficient of the inter-event interval at lag one is not significantly different from 0 (at the 0.99 level):

```
iei_early_cc = np.corrcoef(iei_early[:-1],
                           iei_early[1:])[0,1]*\
                           np.sqrt(len(iei_early)-1)
print(("The inter event interval autocorrelation at lag 1\n"
       "for the pre-stimulation period of neuron 2 from\n"
       "data set e070528citronellal is {0:.4f}").format(iei_early_cc))
```

```
The inter event interval autocorrelation at lag 1
for the pre-stimulation period of neuron 2 from
data set e070528citronellal is 2.3938
```

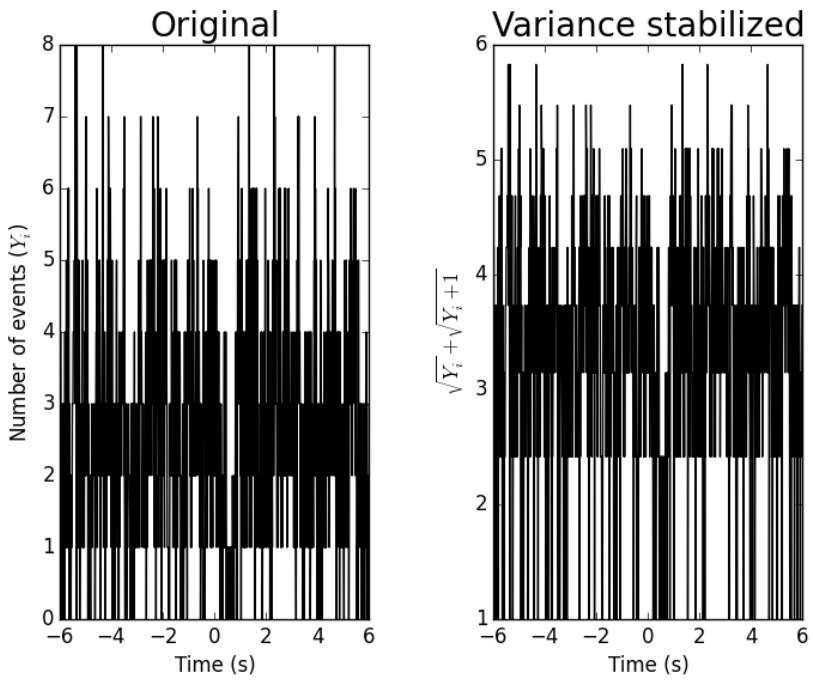
But a plot of the auto-correlation function–with the estimated correlation coeffcient $\hat{\rho}$ is multiplied by the square root of the sample size–up to lag 10 does show some signs of correlations:

```
iei_early_ac = [np.corrcoef(iei_early[:-i],
                            iei_early[i:])[0,1]*\
                np.sqrt(len(iei_early[i:]))
                for i in range(1,11)]
plt.vlines(range(1,11),np.zeros(10),iei_early_ac,lw=2)
plt.grid()
plt.xlim(0,11)
plt.xlabel("Lag",
           fontdict={'fontsize':20})
plt.ylabel(r'$\sqrt{n} \hat{\rho}$',
           fontdict={'fontsize':20})
import scipy.stats as stats
plt.axhline(stats.norm.ppf(0.995),color='blue')
plt.savefig('figs/e070528citronN2ACFPython.png')
plt.close()
'figs/e070528citronN2ACFPython.png'
```



PSTH and variance-stabilized-PSTH figure A figure showing the "counts" and the "stabilized counts" is produced by the following commands:

```
fig = plt.figure()
plt.subplot(121)
citron_spsth_n2.plot(what="counts",ylabel=r'Number of events ($Y_i$)')
plt.title("Original",
          fontdict={'fontsize':20})
plt.subplot(122)
citron_spsth_n2.plot(ylabel=r'$\sqrt{Y_i}+\sqrt{Y_i+1}$')
plt.title("Variance stabilized",
          fontdict={'fontsize':20})
plt.subplots_adjust(wspace=0.4)
plt.savefig('figs/make-n2citron-histos-figure.png')
plt.close()
'figs/make-n2citron-histos-figure.png'
```



2.4.2 Kernel smoothing

The tricube function We start by defining a `tricube_kernel` function:

```
def tricube_kernel(x,bw=1.0):
    ax = np.absolute(x/bw)
    result = np.zeros(x.shape)
    result[ax <= 1] = 70*(1-ax[ax <= 1]**3)**3/81.
    return result
```

Listing 6: `tricube-kernel-definition`

The Nadaraya-Watson estimator We define next a function returning the Nadaraya-Watson estimator at a given point:

```
def NW_Estimator(x,X,Y,
                 kernel = lambda y:
                 tricube_kernel(y,1.0)):
    """Returns the Nadaray-Watson estimator at x, given data X and Y
    using kernel.

    Parameters
    ----------
    x: point at which the estimator is looked for.
    X: abscissa of the observations.
    Y: ordinates of the observations.
    kernel: a univariate 'weight' function.

    Returns
    -------
    The estimated ordinate at x.
    """
    w = kernel(X-x)
    return np.sum(w*Y)/np.sum(w)
```

Listing 7: `Nadaraya-Watson-estimator-definition`

Mallow's Cp score computation We now need a function returning Mallow's C_p score and define a function, `Cp_score`, doing the job:

```
def Cp_score(X,Y,bw = 1.0, kernel = tricube_kernel,sigma2=1):
    """Computes Mallow's Cp score given data X and Y, a bandwidth bw,
    a bivariate function kernel and a variance sigma2.

    Parameters
    ----------
    X: abscissa of the observations.
    Y: ordinates of the observations.
    bw: the bandwidth.
    kernel: a bivariate function taking an ordinate as first parameter
            and a bandwidth as second parameter.
    sigma2: the variance of the ordinates.

    Returns
    -------
    A tuple with the bandwidth, the trace of the smoother and the
    Cp score.
    """
    from numpy.matlib import identity
    L = np.zeros((len(X),len(X)))
    ligne = np.zeros(len(X))
    for i in range(len(X)):
        ligne = kernel(X-X[i], bw)
        L[i,:] = ligne/np.sum(ligne)
    n = len(X)
    trace = np.trace(L)
    if trace == n: return None
    Cp = np.dot(np.dot(Y,(identity(n)-L)),
                np.dot((identity(n)-L),Y).T)[0,0]/n + 2*sigma2*trace/n
    return (bw, trace, Cp)
```

Listing 8: `Cp-score-definition`

In an actual test setting we would use a few kernel bandwidths (1 to 10) in order to have a moderate Bonferroni correction (giving tighter confidence bands); typically we would used multiples of the initial bin width like: 5, 10, 50, 100, 500 leading to:

```
bw_multiplicator = np.array([5,10,50,100,500])
bw_vector = citron_spsth_n2.width*bw_multiplicator
citron_Cp_n2 = np.array([Cp_score(citron_spsth_n2.x,
                                  citron_spsth_n2.y,
                                  bw)
                                for bw in bw_vector])
```

Here, for the sake of illustration, a denser set of bandwidth will also be used:

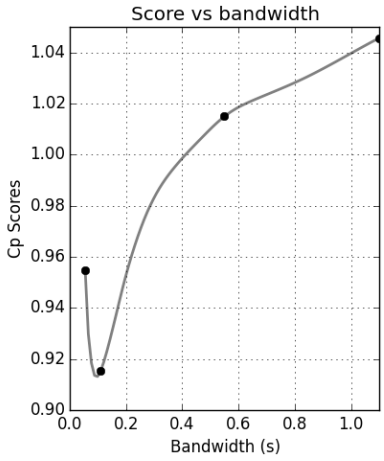
```
bw_multiplicatorDense = np.arange(5,101,1)
bw_vectorDense = citron_spsth_n2.width*bw_multiplicatorDense
citron_CpDense_n2 = np.array([Cp_score(citron_spsth_n2.x,
                                       citron_spsth_n2.y,
                                       bw)
                              for bw in bw_vectorDense])
```

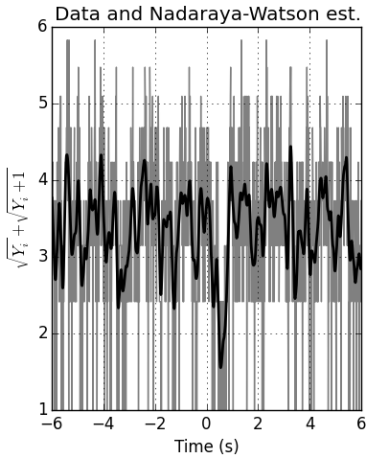
We then extract the bandwidth giving the best (lowest) score and get the corresponding Nadaraya-Watson estimator:

```
bw_best_Cp = bw_vector[np.argmin(citron_Cp_n2[:,2])]
citron_NW_n2 = np.array([NW_Estimator(x,
                                      citron_spsth_n2.x,
                                      citron_spsth_n2.y,
                                      kernel = lambda y:
                                      tricube_kernel(y,
                                                     bw_best_Cp))
                         for x in citron_spsth_n2.x])
```

Figure with Cp score vs bandwidth and smooth estimator Fig. 1 is built with:

```
fig = plt.figure(figsize=(10,5))
plt.subplot(121)
plt.plot(bw_vectorDense,citron_CpDense_n2[:,2],color='grey',lw=2)
plt.plot(bw_vector,citron_Cp_n2[:,2],'ko')
plt.xlabel('Bandwidth (s)')
plt.ylabel('Cp Scores')
plt.title('Score vs bandwidth')
plt.xlim([0,1.1])
plt.ylim([0.9,1.05])
plt.grid(True)
plt.subplot(122)
citron_spsth_n2.plot(ylabel=r'$\sqrt{Y_i}+\sqrt{Y_i+1}$',color='grey')
plt.plot(citron_spsth_n2.x,citron_NW_n2,lw=2,color='black')
plt.title("Data and Nadaraya-Watson est.")
plt.grid(True)
plt.subplots_adjust(wspace=0.4)
plt.savefig('figs/n2citron-Nadaraya-Watson-estimator.png')
plt.close()
'figs/n2citron-Nadaraya-Watson-estimator.png'
```





2.4.3 Confidence set for the smoother

κ_0 We get the value of the integral $IK = \left(\int_a^b K'(t)^2 dt\right)^{1/2}$ appearing in $\kappa_0 \approx (b-a)/h\, IK$ by computing analytically the integral with `sympy`:

```
sx = sy.symbols('sx')
K = 70*(1-sx**3)**3/81 ## symbolic version of the tricube kernel
IK = (sy.sqrt(sy.integrate(sy.diff(K,sx)**2,(sx,0,1))*2)).evalf()
print("The integral of the squared derivative of the kernel is:\n",IK)
```

```
The integral of the squared derivative of the kernel is:
 1.49866250530693
```

We then get the κ_0 for neuron 2:

```
kappa_0_n2 = citron_spsth_n2.support_length*IK/bw_best_Cp
print("The value of kappa_0 is:\n",kappa_0_n2)
```

```
The value of kappa_0 is:
 163.490455124392
```

Getting the constant c of our tube formula We define next a function, `tube_target` returning the "target", that is:

$$2\,(1-\Phi(c)) + \frac{\kappa_0}{\pi} \exp -\frac{c^2}{2} - \alpha\,,$$

```
def tube_target(x,alpha,kappa):
    from scipy.stats import norm
    return 2*(1-norm.cdf(x)) + kappa*np.exp(-x**2/2)/np.pi - alpha
```

Listing 9: `tube-target-definition`

We then get the c values for two α, 0.95 and 0.9 with:

```
from scipy.optimize import brentq
c_p95 = brentq(tube_target,a=3,b=5,args=(0.05/len(bw_vector),kappa_0_n2))
c_p90 = brentq(tube_target,a=2,b=5,args=(0.1/len(bw_vector),kappa_0_n2))
```

Smoothing matrix We define a function returning the smoothing matrix L–a matrix whose $(L)_{i,j}$ element is given by $l_i(t_j)$, where the $l_i()$ are defined in the text and the t_j are the centers of our PSTH bins–:

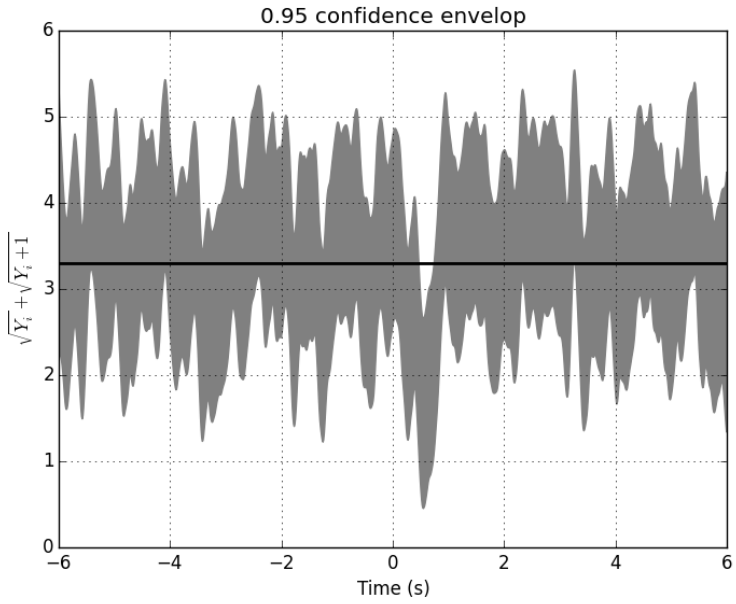
```
def make_L(X,kernel = lambda y: tricube_kernel(y,1.0)):
    result = np.zeros((len(X),len(X)))
    ligne = np.zeros(len(X))
    for i in range(len(X)):
        ligne = kernel(X-X[i])
        result[i,:] = ligne/np.sum(ligne)
    return result
```

Listing 10: `make_L-definition`

```
n2citron_NW_L_best = make_L(citron_spsth_n2.x,
                            kernel = lambda y:
                            tricube_kernel(y,bw_best_Cp))
n2citron_NW_L_best_norm = np.sqrt(np.sum(n2citron_NW_L_best**2,axis=1))
```

Figure of the smooth estimate with the 0.95 confidence set Fig. 2 is simply obtained with:

```
plt.figure()
u = citron_NW_n2+c_p95*n2citron_NW_L_best_norm
l = citron_NW_n2-c_p95*n2citron_NW_L_best_norm
plt.fill_between(citron_spsth_n2.x,u,l,color='grey')
plt.xlim([-6,6])
plt.ylabel(r'$\sqrt{Y_i}+\sqrt{Y_i+1}$')
plt.xlabel('Time (s)')
plt.title("0.95 confidence envelop")
plt.axhline(3.3,lw=2,color='black')
plt.grid(True)
plt.savefig('figs/n2citron-Nadaraya-Watson-Confidence-Bands.png')
plt.close()
'figs/n2citron-Nadaraya-Watson-Confidence-Bands.png'
```



Results of the existings tests Applying the Kolmogorov test and the Anderson-Darling test on the data gives:

```
n2_train = citron_spsth_n2.st + 6
n2_stat = (Kolmogorov_D(n2_train/12),
           AndersonDarling_W2(n2_train/12))
print(("D: {D:.4}, Prob(D): {PD:.4f}\n"
       "W2: {W2:.4f}, Prob(W2): {PW2:.4f}").format(D=n2_stat[0],
                                                  PD=pDsN(n2_stat[0]),
                                                  W2=n2_stat[1],
                                                  PW2=pAD_W2(n2_stat[1])))
```

```
D: 1.278, Prob(D): 0.9238
W2: 1.0392, Prob(W2): 0.6627
```

After jittering and Durbin's transformation we get:

```
n2_train_j = jitter_time(n2_train,(0,12))
n2_train_t = DurbinTransform(n2_train_j,(0,12))
n2t_stat = (Kolmogorov_D(n2_train_t),
            AndersonDarling_W2(n2_train_t))
print(("D: {D:.4}, Prob(D): {PD:.4f}\n"
       "W2: {W2:.4f}, Prob(W2): {PW2:.4f}").format(D=n2t_stat[0],
                                                  PD=pDsN(n2t_stat[0]),
                                                  W2=n2t_stat[1],
                                                  PW2=pAD_W2(n2t_stat[1])))
```

```
D: 1.06, Prob(D): 0.7890
W2: 2.5024, Prob(W2): 0.9506
```

So the critical 0.95 quantile of the Anderson-Darling distribution (2.492) is exceeded but not the 0.99 quantile (3.857).

2.4.4 Define class and methods doing the same job

We can now define a new class, `SmoothStabilizedPSTH`, whose instances contain all the results linked to the kernel smoothing procedure.

```
class SmoothStabilizedPSTH:
    <<docstring_SmoothStabilizedPSTH>>
    <<init_SmoothStabilizedPSTH>>
    <<get_c_SmoothStabilizedPSTH>>
    <<plot_SmoothStabilizedPSTH>>
    <<uplot_SmoothStabilizedPSTH>>
```

Listing 11: `SmoothStabilizedPSTH-definition`

`docstring_SmoothStabilizedPSTH` The `docstring` for the class is defined first:

```
"""Holds a smooth stabilized Peri-Stimuls Time Histogram (PSTH).

Attributes:
  x (1d array): bins' centers.
  y (1d array): stabilized counts.
  n (1d array): actual counts.
  n_stim (scalar): number of trials used to build
    the PSTH.
  width (scalar): bin width.
  stab_method (string): variance stabilization method.
  spontaneous_rate (scalar): spontaneous rate.
  support_length (scalar): length of the PSTH support.
  bandWidthMultipliers (1d array): the bandwidth multipliers
    considered.
  bw_values (1d array): the bandwith values considered.
  trace_values (1d array): traces of the corresponding
    smoothing matrices.
  Cp_values (1d array): Mallow's Cp values.
  bw_best_Cp (scalar): best Cp value.
  NW (1d array): Nadaraya-Watson Estimator with the best
    bandwidth.
  L_best (2d array): smoothing matrix with the best
    bandwidth.
  L_best_norm (1d array): sums of the squared rows of
    L_best.
  kappa_0 (scalar): value of kappa_0.
"""
```

Listing 12: `docstring_SmoothStabilizedPSTH`

`init_SmoothStabilizedPSTH` The class constructor is defined next through its skeleton:

```
def __init__(self,
             sPSTH,
             bandWidthMultipliers = [5,10,50,100],
             sigma2=1):
    <<do-import-and-check-init_SmoothStabilizedPSTH>>
    <<tricube-kernel-definition>>
    <<Nadaraya-Watson-estimator-definition>>
    <<Cp-score-definition>>
    <<make_L-definition>>
    <<get-sPSTH-attributes>>
    <<do-kernel-smoothing>>
    <<prepare-confidence-envelop-computation>>
    <<set-new-attributes>>
```

Listing 13: `init_SmoothStabilizedPSTH`

`do-import-and-check-init_SmoothStabilizedPSTH` The first part of the class constructor consists in importing `numpy` and checking that the type and value of the parameters are correct:

```
import numpy as np
if not isinstance(sPSTH,StabilizedPSTH):
    raise TypeError('sPSTH must be an instance of StabilizedPSTH')
if not np.all(np.array(bandWidthMultipliers)>1):
    raise ValueError('Each element of bandWidthMultipliers must be > 1')
if not sigma2 > 0:
    raise ValueError('sigma2 must be > 0')
```

Listing 14: `do-import-and-check-init_SmoothStabilizedPSTH`

`get-sPSTH-attributes` The attributes of the original `StabilizedPSTH` object are obtained:

```
self.st = sPSTH.st.copy()
self.x = sPSTH.x.copy()
self.y = sPSTH.y.copy()
self.n = sPSTH.n.copy()
self.n_stim = sPSTH.n_stim
self.width = sPSTH.width
self.spontaneous_rate = sPSTH.spontaneous_rate
self.stab_method = sPSTH.stab_method
self.support_length = sPSTH.support_length
```

Listing 15: `get-sPSTH-attributes`

`do-kernel-smoothing` Then the kernel smoothing is performed following the commands sequence previously used:

```
bw_vector = self.width*np.array(bandWidthMultipliers)
Cp_values = np.array([Cp_score(self.x,self.y,bw)
                      for bw in bw_vector])
bw_best_Cp = bw_vector[np.argmin(Cp_values[:,2])]
NW = np.array([NW_Estimator(x,self.x,self.y,
                            kernel = lambda y:
                            tricube_kernel(y,
                                           bw_best_Cp))
               for x in self.x])
```

Listing 16: `do-kernel-smoothing`

`prepare-confidence-envelop-computation` The assignment of the objects required for the confidence band construction is done as before:

```
L_best = make_L(self.x,
                kernel = lambda y:
                tricube_kernel(y,bw_best_Cp))
L_best_norm = np.sqrt(np.sum(L_best**2,axis=1))
IK = 1.49866250530693
kappa_0 = self.support_length*IK/bw_best_Cp
```

Listing 17: `prepare-confidence-envelop-computation`

`set-new-attributes` The new attributes are set:

```
self.bandWidthMultipliers = bandWidthMultipliers
self.bw_values = Cp_values[:,0].copy()
self.trace_values = Cp_values[:,1].copy()
self.Cp_values = Cp_values[:,2].copy()
self.bw_best_Cp = bw_best_Cp
self.NW=NW
self.L_best=L_best
self.L_best_norm=L_best_norm
self.kappa_0 = kappa_0
```

Listing 18: `set-new-attributes`

`get_c_SmoothStabilizedPSTH` Method `get_c` for `SmoothStabilizedPSTH` return the factor c required to build the confidence band at a given level:

```
def get_c(self,alpha=0.05,lower=2,upper=5):
    """Get solution of 2*(1-norm.cdf(x)) +
    kappa*np.exp(-x**2/2)/np.pi - alpha/len(self..bw_vector).

    Parameters
    ----------
    alpha (0 < scalar < 1): the confidence level.
    lower (scalar > 0): the lower starting point of the Brent method.
    upper (scalar > 0): the upper starting point of the Brent method.

    Return
    ------
    The solution (scalar).

    Details
    -------
    The Brent method is used.
    A Bonferroni correction is performed.
    """
    if not 0 < alpha < 1:
        raise ValueError('alpha must be > 0 and < 1')
    if not lower > 0:
        raise ValueError('lower must be > 0')
    if not upper > 0:
        raise ValueError('upper must be > 0')
    <<tube-target-definition>>
    from scipy.optimize import brentq
    return brentq(tube_target,a=lower,b=upper,
                  args=(alpha/len(self.bw_values),self.kappa_0))
```

Listing 19: `get_c_SmoothStabilizedPSTH`

`plot_SmoothStabilizedPSTH` We define now the `plot` method for SmoothStabilizedPSTH instances:

```
def plot(self,what="band",color='black',
         alpha=0.01,lower=2,upper=6,
         ylabel=None,xlabel=None):
    import matplotlib.pyplot as plt
    if not what in ["smooth","band","stab",
                    "Cp vs bandwidth", "Cp vs trace"]:
        msg = 'what should be one of "smooth", "band", '+\
              '"stab", "Cp vs bandwidth", "Cp vs trace"'
        raise ValueError(msg)
    if what in ["smooth","band","stab"]:
        <<plot-smooth-band-stab-SmoothStabilizedPSTH>>
    else:
        <<plot-Cp-SmoothStabilizedPSTH>>
```

Listing 20: `plot_SmoothStabilizedPSTH`

`plot-smooth-band-stab-SmoothStabilizedPSTH` We plot the "smooth", the confidence band or the stabilized data as a function of time:

```
if ylabel is None:
    if self.stab_method == "Freeman-Tukey":
        ylabel = r'$\sqrt{n}+\sqrt{n+1}$'
    elif self.stab_method == "Anscombe":
        ylabel = r'$2 \sqrt{n+3/8}$'
    else:
        ylabel = r'$2 \sqrt{n+1/4}$'
if what == "stab":
    y = self.y
if what == "smooth":
    y = self.NW
if what == "band":
    y = self.NW
    c = self.get_c(alpha,lower,upper)
    u = y + c*self.L_best_norm
    l = y - c*self.L_best_norm
if xlabel is None:
    xlabel = "Time (s)"
if what in ["stab","smooth"]:
    plt.plot(self.x,y,color=color)
else:
    plt.fill_between(self.x,u,l,color=color)
plt.xlabel(xlabel)
plt.ylabel(ylabel)
```

Listing 21: `plot-smooth-band-stab-SmoothStabilizedPSTH`

`plot-Cp-SmoothStabilizedPSTH` We plot Mallow's Cp value as a function of the bandwidth of the smoothing matrix trace:

```
y = self.Cp_values
if ylabel is None:
    ylabel = "Cp"
if what == "Cp vs bandwidth":
    X = self.bw_values
    if xlabel is None:
        xlabel = "Bandwidth (s)"
else:
    X = self.trace_values
    if xlabel is None:
        xlabel = "Smoother trace"
plt.plot(X,y,color=color)
plt.xlabel(xlabel)
plt.ylabel(ylabel)
```

Listing 22: `plot-Cp-SmoothStabilizedPSTH`

`uplot_SmoothStabilizedPSTH` `uplot` is another `plot` method for `SmoothStabilizedPSTH` instances where the variance stabilization is "undone", that is, the plot of a classical PSTH is produced. When a confidence band is drawn there is a potential caveat if the lower bound of the band has a lower value than the transformed / stabilized value of the lowest possible count, 0. In those cases, the inverse value of the lower bound will the set to the inverse of the transformed value of 0.

```
def uplot(self,what="band",color='black',
          alpha=0.01,lower=2,upper=6,
          ylabel=None,xlabel=None):
    import matplotlib.pyplot as plt
    if not what in ["smooth","band","stab"]:
        msg = 'what should be one of "smooth", "band", "stab"'
        raise ValueError(msg)
    if ylabel is None:
        ylabel = "Frequency (Hz)"
    if xlabel is None:
        xlabel = "Time (s)"
    if what == "stab":
        y = self.y
    else:
        y = self.NW
    if what == "band":
        c = self.get_c(alpha,lower,upper)
        u = y + c*self.L_best_norm
        l = y - c*self.L_best_norm
    if self.stab_method == "Freeman-Tukey":
        def InvFct(y):
            y = np.maximum(y,1)
            return ((y**2-1)/2./y)**2/self.n_stim/self.width
    if self.stab_method == "Anscombe":
        def InvFct(y):
            y = np.maximum(y,2*np.sqrt(3/8.))
            return (y**2/4. + np.sqrt(1.5)/4./y - 11/8./y**2 -\
                    1/8.)/self.n_stim/self.width
    if self.stab_method == "Brown et al":
        def InvFct(y):
            y = np.maximum(y,1)
            return (y**2/4.-0.25)/self.n_stim/self.width
    y = InvFct(y)
    if what in ["stab","smooth"]:
        plt.plot(self.x,y,color=color)
    else:
        u = InvFct(u)
        l = InvFct(l)
        plt.fill_between(self.x,u,l,color=color)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
```

Listing 23: `uplot_SmoothStabilizedPSTH`

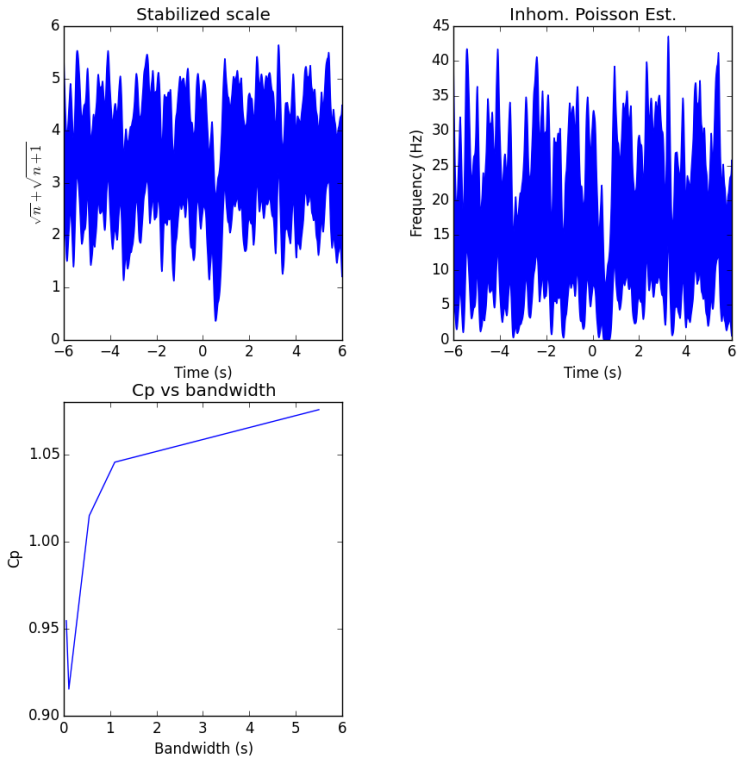
Tests Few tests to make sure we get what we got before...

```
citron_sspsth_n2 = SmoothStabilizedPSTH(citron_spsth_n2,[5,10,50,100,500])
```

The next figure shows the 99% confidence bands (top left), the Cp values

as a function of the bandwidth (bottom left) and the estimated inhomogeneous Poisson intensity with 95% confidence bands (top right):

```
fig = plt.figure(figsize=(10,10))
plt.subplot(221)
citron_sspsth_n2.plot(color='blue',alpha=0.01)
plt.title('Stabilized scale')
plt.subplot(222)
citron_sspsth_n2.uplot(color='blue',alpha=0.05)
plt.title('Inhom. Poisson Est.')
plt.subplot(223)
citron_sspsth_n2.plot(color='blue',what="Cp vs bandwidth")
plt.title('Cp vs bandwidth')
plt.subplots_adjust(wspace=0.4)
plt.savefig('figs/test-SmoothStabilizedPSTH-fig.png')
plt.close()
'figs/test-SmoothStabilizedPSTH-fig.png'
```



2.5 Systematic analysis

We can now analyze all the odor responses of the data set in the same way, building 99% confidence bands using 5 seconds before the stimulus onset and 6 seconds after it (the longest compromise among our data sets).

2.5.1 Experiment e060817

We get the spontaneous discharge rates of the three neurons of experiment e060817:

```
e060817_spont_nu = [len(f["e060817/Neuron"+str(i)+"/spont"])/60
                    for i in range(1,4)]
print("The spontaneous discharge rates are:")
for i in range(len(e060817_spont_nu)):
    print("  Neuron {0}: {1:.2f} (Hz)".format(i+1,e060817_spont_nu[i]))
```

```
The spontaneous discharge rates are:
  Neuron 1: 8.82 (Hz)
  Neuron 2: 20.48 (Hz)
  Neuron 3: 13.02 (Hz)
```

We create next `StabilizedPSTH` instances corresponding to the citronellal responses of each neuron as well as the `SmoothStabilizedPSTH`:

```
citron_onset = f["e060817/Neuron1/citronellal/stimOnset"][...][0]
e060817citron = [[f[y][...] for y in
                  ["e060817/Neuron"+str(i)+"/citronellal/stim"+str(x)
                   for x in range(1,21)]]
                 for i in range(1,4)]
e060817citron_spsth = [StabilizedPSTH(train_list,
                                      spontaneous_rate=spont_nu,
                                      onset=citron_onset,
                                      region = [-5,6])
                       for train_list,spont_nu in zip(e060817citron,
                                                      e060817_spont_nu)]
e060817citron_sspsth = [SmoothStabilizedPSTH(spsth)
                        for spsth in e060817citron_spsth]
```

The terpineol and mixture responses are processed with:

```
terpi_onset = f["e060817/Neuron1/terpineol/stimOnset"][...][0]
e060817terpi = [[f[y][...] for y in
                 ["e060817/Neuron"+str(i)+"/terpineol/stim"+str(x)
                  for x in range(1,21)]]
                for i in range(1,4)]
e060817terpi_spsth = [StabilizedPSTH(train_list,
                                     spontaneous_rate=spont_nu,
                                     onset=terpi_onset,
                                     region = [-5,6])
                      for train_list,spont_nu in zip(e060817terpi,
                                                     e060817_spont_nu)]
e060817terpi_sspsth = [SmoothStabilizedPSTH(spsth)
                       for spsth in e060817terpi_spsth]
mix_onset = f["e060817/Neuron1/mixture/stimOnset"][...][0]
e060817mix = [[f[y][...] for y in
               ["e060817/Neuron"+str(i)+"/mixture/stim"+str(x)
                for x in range(1,21)]]
              for i in range(1,4)]
e060817mix_spsth = [StabilizedPSTH(train_list,
                                   spontaneous_rate=spont_nu,
                                   onset=mix_onset,
                                   region = [-5,6])
                      for train_list,spont_nu in zip(e060817mix,
                                                     e060817_spont_nu)]
e060817mix_sspsth = [SmoothStabilizedPSTH(spsth)
                     for spsth in e060817mix_spsth]
```

2.5.2 Experiment e060824

This data set contains only two neurons and a single odor response (to citral). The analysis is done with:

```
e060824_spont_nu = [len(f["e060824/Neuron"+str(i)+"/spont"])/59
                    for i in range(1,3)]
citral_onset = f["e060824/Neuron1/citral/stimOnset"][...][0]
e060824citral = [[f[y][...] for y in
                  ["e060824/Neuron"+str(i)+"/citral/stim"+str(x)
                   for x in range(1,21)]]
                 for i in range(1,3)]
e060824citral_spsth = [StabilizedPSTH(train_list,
                                      spontaneous_rate=spont_nu,
                                      onset=citral_onset,
                                      region = [-5,6])
                       for train_list,spont_nu in zip(e060824citral,
                                                      e060824_spont_nu)]
e060824citral_sspsth = [SmoothStabilizedPSTH(spsth)
                        for spsth in e060824citral_spsth]
```

2.5.3 Experiment e060517

This data set contains the responses of three neurons to ionon:

```
e060517_spont_nu = [len(f["e060517/Neuron"+str(i)+"/spont"])/61
                    for i in range(1,4)]
ionon_onset = f["e060517/Neuron1/ionon/stimOnset"][...][0]
e060517ionon = [[f[y][...] for y in
                 ["e060517/Neuron"+str(i)+"/ionon/stim"+str(x)
                  for x in range(1,20)]]
                for i in range(1,4)]
e060517ionon_spsth = [StabilizedPSTH(train_list,
                                     spontaneous_rate=spont_nu,
                                     onset=ionon_onset,
                                     region = [-5,6])
                      for train_list,spont_nu in zip(e060517ionon,
                                                     e060517_spont_nu)]
e060517ionon_sspsth = [SmoothStabilizedPSTH(spsth)
                       for spsth in e060517ionon_spsth]
```

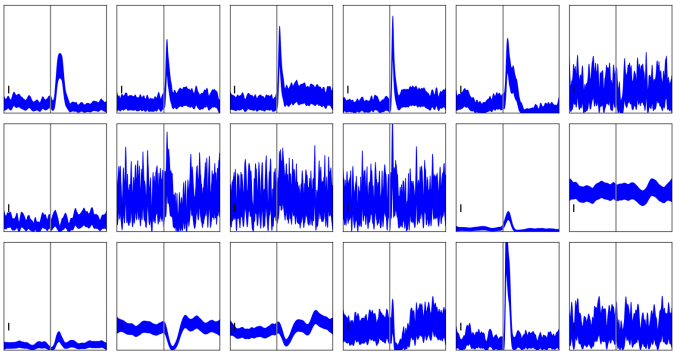
2.5.4 Experiment e070528

This data set contains the responses of four neurons to citronellal:

```
e070528_spont_nu = [len(f["e070528/Neuron"+str(i)+"/spont"])/60
                    for i in range(1,5)]
citron_onset = f["e070528/Neuron1/citronellal/stimOnset"][...][0]
e070528citron = [[f[y][...] for y in
                  ["e070528/Neuron"+str(i)+"/citronellal/stim"+str(x)
                   for x in range(1,16)]]
                 for i in range(1,5)]
e070528citron_spsth = [StabilizedPSTH(train_list,
                                      spontaneous_rate=spont_nu,
                                      onset=citron_onset,
                                      region = [-5,6])
                       for train_list,spont_nu in zip(e070528citron,
                                                      e070528_spont_nu)]
e070528citron_sspsth = [SmoothStabilizedPSTH(spsth)
                        for spsth in e070528citron_spsth]
```

2.5.5 A new version of Fig. 8 of Pouzat and Chaffiol (2009)

We can now make a new version of Fig. 8 of Pouzat and Chaffiol (2009) with 99% confidence bands instead of 95% pointwise confidence intervals using the "natural" scale, the one on which the variance has been stabilized:



2.6 Testing identity

2.6.1 Boundary crossing probability

Background We are going to need the probability for a canonical Brownian motion to cross a boundary whose equation is $a + b\sqrt{t}$ between time 0 and time 1. To this end we use the results of Loader and Deely (1987) that can be summarized as follows, writing $G(t)$ the CDF of the first passage time, $g(t)$ the corresponding density and $c(t)$ a *continuous* boundary. We can choose a function $b(t)$, then G is solution of the following Volterra integral equation:

$$F(t) = \int_0^t K(t,u)dG(u)\,,$$

where

$$F(t) = \Phi\left(-\frac{c(t)}{\sqrt{t}}\right) + \exp\left(-2b(t)\left(c(t) - tb(t)\right)\right)\Phi\left(\frac{-c(t)+2tb(t)}{\sqrt{t}}\right)$$

and

$$\begin{aligned} K(t,u) &= \Phi\left(-\tfrac{c(u)-c(t)}{\sqrt{t-u}}\right) + \\ &\quad \exp\left(-2b(t)\left(c(t) - c(u) - (t-u)b(t)\right)\right)\Phi\left(\tfrac{c(u)-c(t)+2(t-u)b(t)}{\sqrt{t-u}}\right)\,. \end{aligned}$$

We now take $0 = t_0 < t_1 < \cdots < t_n = t$ with $t_j = jh$ for some $h > 0$ and we set $t_{j-1/2} = (t_j + t_{j-1})/2$ a discretized version of our Volterra equation is then given by the *mid-point method*:

$$F(t_j) = \sum_{i=1}^{j} K(t_j, t_{i-1/2})\Delta_i \quad j = 1, \ldots, n\,,$$

where $\Delta_i = G(t_i) - G(t_{i-1})$ and since this linear system is lower triangular we get:

$$\Delta_j = \left(F(t_j) - \sum_{i=1}^{j-1} K(t_j, t_{i-1/2}) \Delta_i \right) / K(t_j, t_{j-1/2}) \quad j = 1, \ldots, n\,.$$

Assuming that $c'(t)$ exists for all $t > 0$ and setting $L(t,u) = \partial K(t,u)/\partial u$,

$$\begin{array}{rcl} G_L(t_1) & = & F(t_1) \\ G_L(t_n) & = & F(t_n) + \sum_{j=1}^{n-1} G_L(t_j) \left[K(t_n, t_{j+1} - K(t_n, t_j) \right] \quad n = 2, \ldots \end{array}$$

and

$$\begin{array}{rcl} G_U(t_1) & = & F(t_1)/K(t_1, t_0) \\ G_U(t_n) & = & \left\{ F(t_n) + \sum_{j=1}^{n-1} G_U(t_j) \left[K(t_n, t_j - K(t_n, t_{j-1}) \right] \right\} / K(t_n, t_{n-1}) \\ & & n = 2, \ldots \end{array}$$

Loader and Deely (1987) show that if $L(t,u) \geq 0$ for $u < t$ then

$$G_L(t_n) \leq G(t_n) \leq G_U(t_n) \quad n = 1, 2, \ldots$$

`Python` code We present next a direct implementation of this algorithm in `Python`. We start with the `docstring` (user documentation of the function):

```
"""Probabilty for a canonical Brownian motion to cross a boundary
defined by the continous function c_fct beween 0 and 1.

Parameters
----------
c_fct: a continuous function of a single variable defining the
       boundary.
b_fct: an accessory function helping the convergence, the
       derivative of c_fct is a good default choice.
bounds: a Boolean variable, if True (default) lower and upper
        bounds for the probability are returned.

Returns
-------
The probability if bounds is False or a tuple with the lower bound
the probability and the upper bound.

Details
-------
Bounds calculation uses Eq. 3.6 and 3.7 p 102 of Loader and Deely
(1987) J Statist Comput Simul 27: 95-105, and some conditions on
the partial derivative of the Kernel appearing in the Volterra
integral equation are supposed to be met."""
```

In the actual `G_at_1_with_bounds` definition below, `«G_at_1_with_bounds-docstring»` should be replaced by the code above. We then define a univariate function `F` corresponding the function F above. This function needs to have access to the `norm` class of `scipy.stats` and to have access to two functions `c_fct` and `b_fct` corresponding respectively to c and b:

```
def F(t):
    c_t = c_fct(t)
    b_t = b_fct(t)
    term1 = norm.cdf(-c_t/np.sqrt(t))
    factorA =  np.exp(-2*b_t*(c_t-t*b_t))
    factorB = norm.cdf((-c_t+2*t*b_t)/np.sqrt(t))
    return term1 + factorA*factorB
```

In the actual `G_at_1_with_bounds` definition below, `«F-definition»` is meant to be replaced by the above code. We define next a bivariate function `K` corresponding to K above and requiring the same functions `c_fct` and `b_fct` as `F`. This function implicitely assumes that $c(u) - c(t)$ falls to 0 faster than $\sqrt{t-u}$ when $t > 0$ and $u \to t$:

```
def K(t,u):
    if t == u:
        return 1.0
    c_t = c_fct(t)
    c_u = c_fct(u)
    b_t = b_fct(t)
    term1 = norm.cdf((c_u-c_t)/np.sqrt(t-u))
    factorA =  np.exp(-2*b_t*(c_t-c_u-(t-u)*b_t))
    factorB = norm.cdf((c_u-c_t+2*(t-u)*b_t)/np.sqrt(t-u))
    return term1 + factorA*factorB
```

We now define the user function `G_at_1_with_bounds`:

```
def G_at_1_with_bounds(c_fct,b_fct,n,bounds=True):
    <<G_at_1_with_bounds-docstring>>
    from scipy.stats import norm
    <<F-definition>>
    <<K-definition>>
    t_v = np.linspace(0,1,n+1)
    t_v_half = (t_v[1:]+t_v[:-1])*0.5
    Delta = np.zeros((n))
    if bounds:
        G_L = np.zeros((n))
        G_U = np.zeros((n))
        G_L[0] = F(t_v[1])
        G_U[0] = F(t_v[1])/K(t_v[1],t_v[0])
    Delta[0] = F(t_v[1])/K(t_v[1],t_v_half[0])
    for j in range(1,n):
        term1 = F(t_v[j+1])
        factor1 = Delta[:j]
        factor2 = [K(t_v[j+1],t) for t in t_v_half[:j]]
        term2 = np.sum(factor1*np.array(factor2))
        divisor = K(t_v[j+1],t_v_half[j])
        Delta[j] = (term1-term2)/divisor
        if bounds:
            factor2 = np.diff(np.array([K(t_v[j+1],t)
                                        for t in t_v[:(j+2)]]))
            G_L[j] = term1 + np.sum(G_L[:j]*factor2[1:])
            G_U[j] = (term1 +
                      np.sum(G_U[:j]*factor2[:-1]))/K(t_v[j+1],t_v[j])
    if bounds:
        return (G_L[n-1],np.sum(Delta),G_U[n-1])
    else:
        return np.sum(Delta)
```

Test against Loader and Deely reported results We can check our code against the results reported in Table II p 104 of Loader and Deely (1987), starting with the first "column" of the upper part of the table:

```
LD87tableIIaa = [G_at_1_with_bounds(lambda x: np.sqrt(1+x),
                                    lambda x: 0.5/np.sqrt(1+x),n)
                                    for n in [8,16,32,64,128]]

[[str(round(x[0],5)),str(round(x[2],5))] for x in LD87tableIIaa]
```

```
Bounds for G(t) when the boundary is (1+t)^(1/2) using
b(t)=0.5/(1+t)^(1/2):
  n  L & D Gl(1) Present Gl(1)  L & D Gu(1)  Present Gu(1)
  8      0.19524       0.19524      0.19690        0.19690
 16      0.19560       0.19560      0.19643        0.19643
 32      0.19580       0.19580      0.19621        0.19621
 64      0.19590       0.19590      0.19610        0.19610
```

```
128      0.19595       0.19595       0.19605        0.19605
```

Parameters of the "square root boundary" We are going to consider a simple boundary leading to an almost minimal surface (Kendall, Marin, and Robert 2007), that is a "square root boundary" $a+b\sqrt{t}$. Kendall, Marin, and Robert (2007) report in their Table 1 that $a = 0.3$ and $b = 2.35$ give a 0.95 "coverage probability". Partitioning [0,1] in 256 equal parts we get:

```
print("""
Lower bound: {G[0]:.6f}
Best guess:  {G[1]:.6f}
Upper bound: {G[2]:.6f}
""".format(G=G_at_1_with_bounds(lambda x: 0.3+2.35*np.sqrt(x),
                                lambda x: 0.5*2.35/np.sqrt(x),256)))
```

```
Lower bound: 0.024756
Best guess:  0.024864
Upper bound: 0.024975
```

We can refine these values by defining first a function returning a target function (to optimize later) with:

```
def mk_boundary_target(alpha=0.05,n=128):
    target = 0.5*alpha
    def b_target(log_x):
        a = np.exp(log_x[0])
        b = np.exp(log_x[1])
        return (target -
                G_at_1_with_bounds(lambda y: a+b*np.sqrt(y),
                                   lambda y: 0.5*b/np.sqrt(y),n,
                                   False))**2
    return b_target
```

We use our "target making" function:

```
b95target = mk_boundary_target(alpha=0.05,n=128)
```

And we refine our parameters:

```
from scipy.optimize import minimize
b95 = minimize(b95target,[np.log(0.3),np.log(2.35)],
               method='BFGS',options={'disp': True})
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 2
         Function evaluations: 16
         Gradient evaluations: 4
```

```
a_95,b_95 = (np.exp(x) for x in b95.x)
print("""
Using a = {a:.6f} and b = {b:.6f} we get:
Lower bound: {G[0]:.6f}
Best guess:  {G[1]:.6f}
Upper bound: {G[2]:.6f}
""".format(a=a_95,b=b_95,
           G=G_at_1_with_bounds(lambda x: a_95+b_95*np.sqrt(x),
                                lambda x: 0.5*b_95/np.sqrt(x),256)))
```

```
Using a = 0.299957 and b = 2.348404 we get:
Lower bound: 0.024863
Best guess:  0.024971
Upper bound: 0.025083
```

We made a systematic estimation of the parameters a and b of the square root boundary for coverage probabilities going from 0.9 to 0.99. To that end we defined a "square root boundary tailored version" of `G_at_1_with_bounds` that makes a much better use of the vectorization allowed (en encouraged) by `Python`. We do not give the code in this document but it is fully disclosed in its source file.

We end up with the following coefficient table:

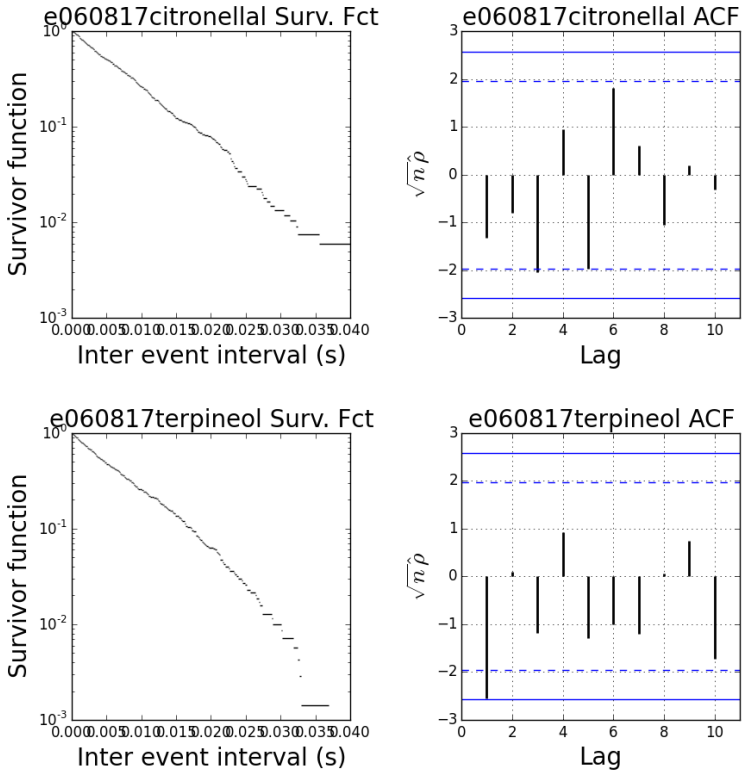
```
Square root boundary, a+b*sqrt(t), coefficients as a function
of the probability for a Brownian motion process to be totally
within the domain up to time 1.
---------------------------------------------------------------
Prob         a         b
0.90  0.291810  2.077198
0.91  0.293235  2.120344
0.92  0.294731  2.167435
0.93  0.296332  2.220010
0.94  0.298058  2.279445
0.95  0.299958  2.348443
0.96  0.302124  2.429348
0.97  0.304680  2.531266
0.98  0.307846  2.668233
0.99  0.312456  2.890606
```

Back to the analysis of the data set We have already built the citronellal and terpineol PSTHs of neuron 1. We start by checking that during the pre-stimulation period the aggregated processes have the properties of an homogeneous Poisson process.

```
e060817citron_n1 = e060817citron_spsth[0].st
e060817citron_e1 = e060817citron_n1[e060817citron_n1<0]+5
e060817citron_j1 = jitter_time(e060817citron_e1,(0,5))
e060817citron_t1 = DurbinTransform(e060817citron_j1,(0,5))
e060817terpi_n1 = e060817terpi_spsth[0].st
e060817terpi_e1 = e060817terpi_n1[e060817terpi_n1<0]+5
e060817terpi_j1 = jitter_time(e060817terpi_e1,(0,5))
e060817terpi_t1 = DurbinTransform(e060817terpi_j1,(0,5))
e060817comp_test = {"cDo":Kolmogorov_D(e060817citron_e1/5),
                    "cW2o":AndersonDarling_W2(e060817citron_e1/5),
                    "cDt":Kolmogorov_D(e060817citron_t1),
                    "cW2t":AndersonDarling_W2(e060817citron_t1),
                    "tDo":Kolmogorov_D(e060817terpi_e1/5),
                    "tW2o":AndersonDarling_W2(e060817terpi_e1/5),
                    "tDt":Kolmogorov_D(e060817terpi_t1),
                    "tW2t":AndersonDarling_W2(e060817terpi_t1),
                    "TDo":"D original",
                    "TW2o":"W2 original",
                    "TDt":"D transformed",
                    "TW2t":"W2 transformed",
                    "line1":"citronellal",
                    "line2":"terpineol"}
e060817comp_out = ("               {TDo:>14} {TW2o:>14}"
                   " {TDt:>14} {TW2t:>14}\n"
                   "{line1:<14} {cDo:>14.4f} {cW2o:>14.4f}"
                   " {cDt:>14.4f} {cW2t:>14.4f}\n"
                   "{line2:<14} {tDo:>14.4f} {tW2o:>14.4f}"
                   " {tDt:>14.4f} {tW2t:>14.4f}").format(**e060817comp_test)
print(e060817comp_out)
```

```
                D original    W2 original  D transformed W2 transformed
citronellal         0.8559         0.8342         0.6475         0.6081
terpineol           0.5494         0.3413         1.0007         1.3110
```

The log-survivor function as well as the auto-correlation function of the inter event intervals with the two stimulations are :

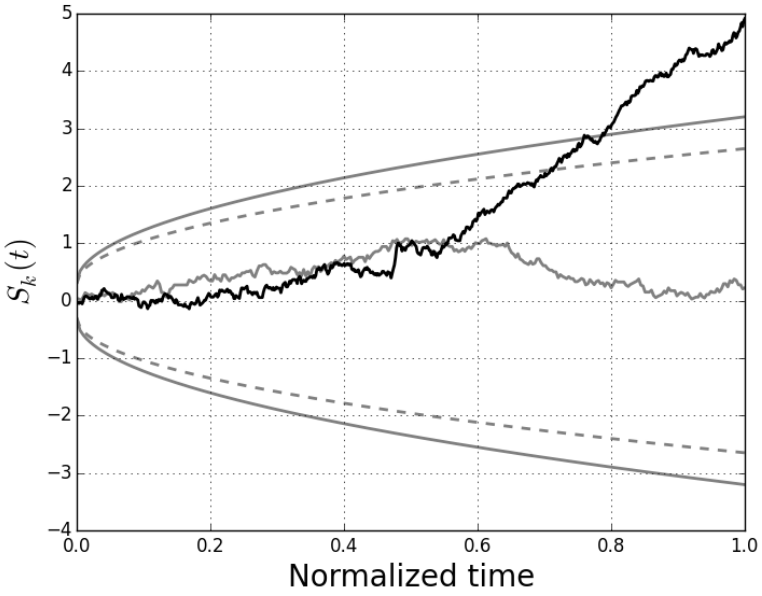


We now want to build `stabilizedPSTH` instances corresponding to the even and odd terpineol stimulations:

```
n1terpiEven = e060817terpi[0][0:20:2]
n1terpiEven_spsth = StabilizedPSTH(n1terpiEven,
                                   spontaneous_rate=e060817_spont_nu[0],
                                   onset=terpi_onset,
                                   region = [-5,6])
n1terpiOdd = e060817terpi[0][1:20:2]
n1terpiOdd_spsth = StabilizedPSTH(n1terpiOdd,
                                  spontaneous_rate=e060817_spont_nu[0],
                                  onset=terpi_onset,
                                  region = [-5,6])
```

```
def c95(x): return sqrt_coef[5][1]+sqrt_coef[5][2]*np.sqrt(x)

def c99(x): return sqrt_coef[9][1]+sqrt_coef[9][2]*np.sqrt(x)
```

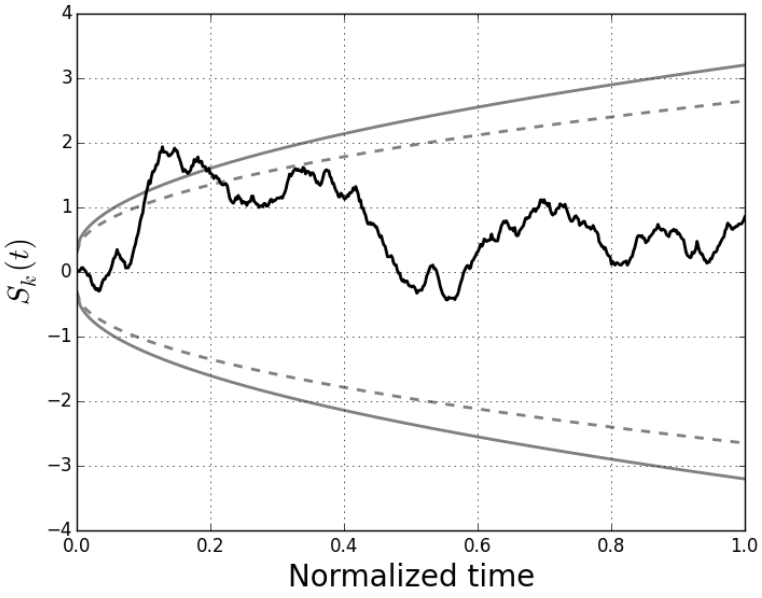
We can now make Fig. 5 with:



We now consider the citronellal response of neuron 2 from data set `e070528`. The idea here is to compare the 6 seconds prior to stimulus presentation with the 6 seconds after. So we start by building 2 `StabilizedPSTH` instance corresponding to the two parts:

```
citron_onset = f["e070528/Neuron2/citronellal/stimOnset"][...][0]
citron_spsth_n2_before = StabilizedPSTH(e070528citron[1],
                                        spontaneous_rate=e070528_spont_nu[1],
                                        region = [-6,0],
                                        onset=citron_onset)
citron_spsth_n2_after = StabilizedPSTH(e070528citron[1],
                                       spontaneous_rate=e070528_spont_nu[1],
                                       region = [0,6],
                                       onset=citron_onset)
Y_before = citron_spsth_n2_before.y
Y_after = citron_spsth_n2_after.y
Y_diff = (Y_before-Y_after)/np.sqrt(2)
Y_NCM = np.cumsum(Y_diff)/np.sqrt(len(Y_diff))
X_NCM = np.arange(1,len(Y_diff)+1)/len(Y_diff)
```

The test figure is obtained with:



2.7 Simulation study

We want to estimate the coverage probability of our "Brownian domains" as a function of the sample size. We are going to use a Monte Carlo simulation to do that for each of our nine sets of square root boundary coefficients. To that end we define first a function carrying out the simulations at a given sample size–the normal (pseudo-) random numbers are generated with the function `normal` from module `numpy.random`, examination of the source code shows that these normal random numbers are generated with the Box-Muller algorithm–:

```
def inside_domain(sample_size,
                  n_rep=100000,
                  coeff_list=sqrt_coef):
    """Computes a 95% confidence interval for the 'coverage
    probability' of each square-root boundary defined in the list
    coeff_list for a given sample size using n_rep Monte Carlo
    replicates.

    Parameters
    ----------
    sample_size: an integer, the sample size.
    n_rep: an integer, the number of MC replicates.
    coeff_list: a list of lists. Each sub list should contain the
                coefficient a and b in its second and third elements,
                the boundary being defined by: a + b*sqrt(t).

    Returns
    -------
    A list of tuple, each subtuple contains the extremes of an
    Agresti-Coull 95% CI as defined by Brown et al (2001) Statistical
    Science 16:101-117. There is on list element for each element of
    coeff_list."""
    from numpy.random import normal
    t_v = np.arange(1,(sample_size+1))/float(sample_size)
    b_list = [coeff[1]+coeff[2]*np.sqrt(t_v) for coeff in coeff_list]
    total_v = np.zeros((len(coeff_list)))
    for i in range(n_rep):
        s = np.cumsum(normal(size=sample_size))/np.sqrt(sample_size)
        inside = [np.all(s.__le__(B)) and np.all(s.__ge__(-B))
                  for B in b_list]
        total_v += np.array(inside,dtype=int)
    proba = [(T+2)/(n_rep+4.) for T in total_v]
    res = [(p - 2*np.sqrt(p*(1-p)/(n_rep+4.)),
            p + 2*np.sqrt(p*(1-p)/(n_rep+4.)))
            for p in proba]
    return res
```

We then use this function to get the empirical coverage probabilities in a range of sample sizes:

```
np.random.seed(20110928)
samp_size_list = [25,50,75,100,250,500,750,1000,2500,5000,7500,10000]
emp_CP = [inside_domain(samp_size)
          for samp_size in samp_size_list]
```

Empirical coverage probabilities (presented as lower and upper bounds of 95% confidence intervals) tabulated as a function of the nominal coverage probability (rows) and of the sample size (columns).

	25	50	75	100	250	500	750	1000	2500	5000	7500	10000
0.99 up	0.995	0.993	0.993	0.993	0.992	0.992	0.991	0.991	0.991	0.991	0.991	0.991
0.99 low	0.994	0.992	0.992	0.992	0.991	0.991	0.990	0.990	0.990	0.990	0.990	0.989
0.98 up	0.988	0.987	0.986	0.985	0.984	0.984	0.982	0.982	0.981	0.981	0.981	0.981
0.98 low	0.987	0.985	0.984	0.983	0.982	0.982	0.981	0.980	0.979	0.980	0.980	0.979
0.97 up	0.983	0.980	0.979	0.977	0.975	0.975	0.973	0.973	0.972	0.972	0.972	0.972
0.97 low	0.981	0.978	0.977	0.975	0.973	0.973	0.971	0.971	0.970	0.970	0.970	0.970
0.96 up	0.976	0.973	0.971	0.970	0.967	0.965	0.964	0.964	0.962	0.962	0.962	0.962
0.96 low	0.974	0.971	0.969	0.967	0.965	0.963	0.961	0.961	0.960	0.960	0.959	0.959
0.95 up	0.970	0.966	0.964	0.962	0.959	0.957	0.955	0.954	0.953	0.953	0.953	0.952
0.95 low	0.968	0.963	0.961	0.959	0.956	0.954	0.952	0.951	0.950	0.950	0.950	0.949
0.94 up	0.963	0.958	0.956	0.954	0.950	0.948	0.946	0.945	0.943	0.943	0.943	0.942
0.94 low	0.961	0.956	0.953	0.951	0.947	0.945	0.943	0.942	0.940	0.940	0.940	0.939
0.93 up	0.957	0.951	0.948	0.946	0.942	0.939	0.937	0.936	0.934	0.934	0.933	0.933
0.93 low	0.954	0.948	0.945	0.943	0.939	0.935	0.933	0.933	0.931	0.931	0.930	0.930
0.92 up	0.951	0.944	0.939	0.937	0.933	0.929	0.927	0.927	0.924	0.925	0.924	0.923
0.92 low	0.948	0.941	0.936	0.934	0.930	0.926	0.924	0.924	0.921	0.922	0.920	0.919
0.91 up	0.944	0.937	0.931	0.929	0.925	0.920	0.918	0.918	0.915	0.915	0.914	0.913
0.91 low	0.941	0.933	0.928	0.926	0.921	0.917	0.914	0.915	0.912	0.911	0.910	0.909
0.90 up	0.938	0.929	0.923	0.921	0.916	0.911	0.909	0.909	0.905	0.905	0.904	0.903
0.90 low	0.935	0.925	0.920	0.917	0.912	0.908	0.905	0.905	0.902	0.902	0.901	0.899

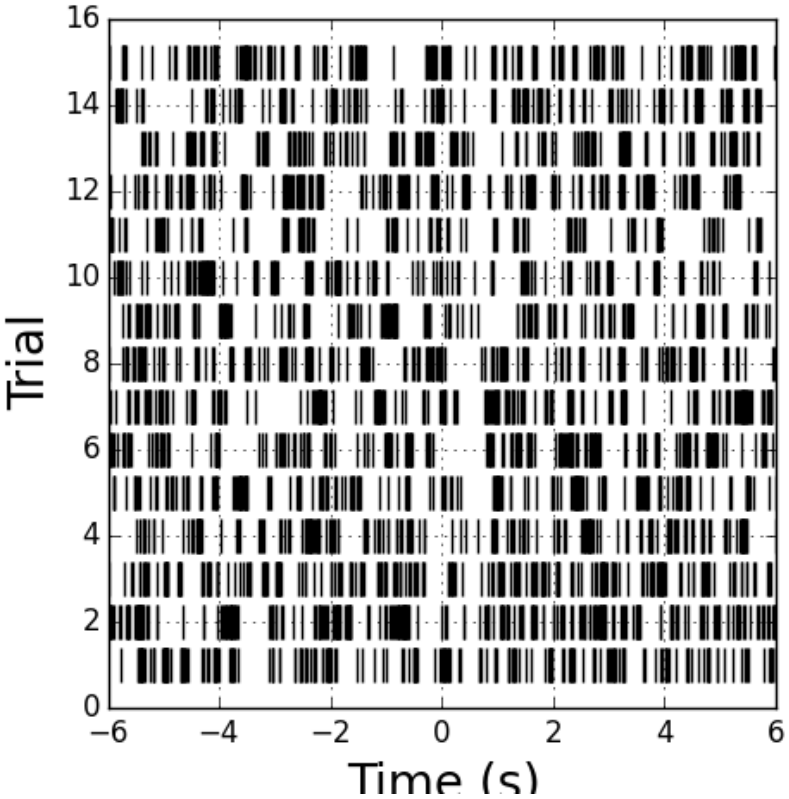
We get the results shown on Table 2.

2.8 Raster plots

```
def raster_plot(train_list,
                stim_onset=None,
                color = 'black'):
    """Create a raster plot.

    Parameters
    ----------
    train_list: a list of spike trains (1d vector with strictly
      increasing elements).
    stim_onst: a number giving the time of stimulus onset. If
      specificied, the time are realigned such that the stimulus
      comes at 0.
    color: the color of the ticks representing the spikes.

    Side effect:
    A raster plot is created.
    """
    import numpy as np
    import matplotlib.pyplot as plt
    if stim_onset is None:
        stim_onset = 0
    for idx,trial in enumerate(train_list):
        plt.vlines(trial-stim_onset,
                   idx+0.6,idx+1.4,
                   color=color)
    plt.ylim(0.5,len(train_list))
```

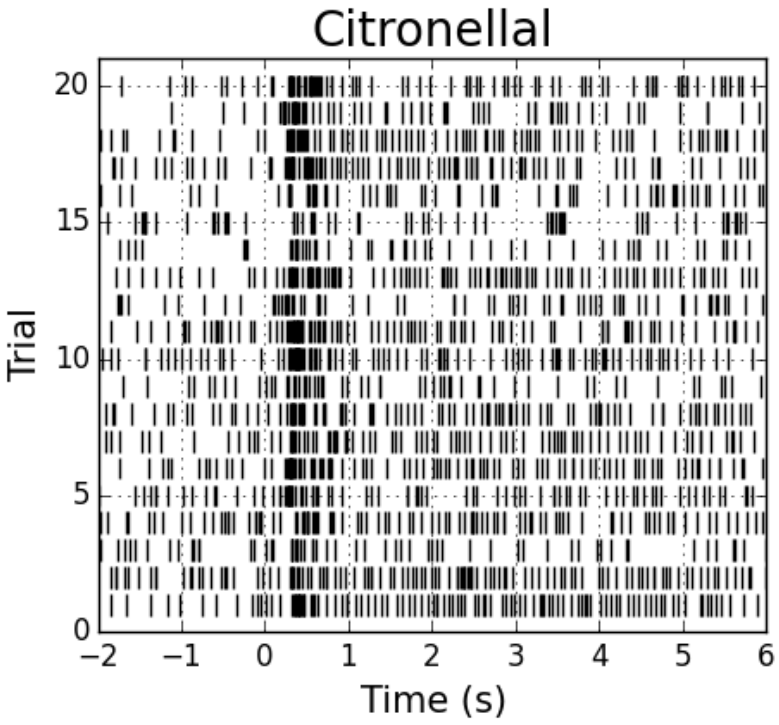
The first raster plot is then obtained with:

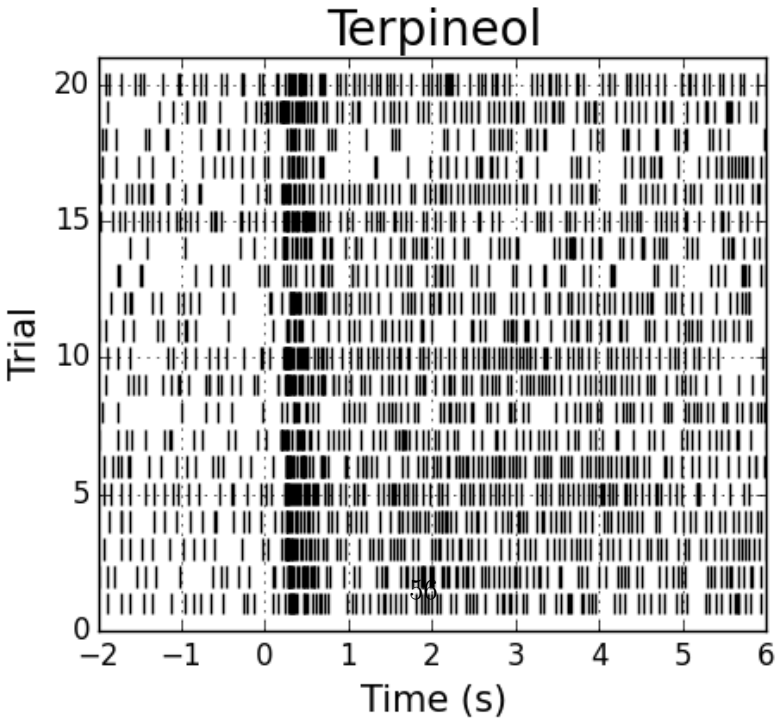
```
citron_onset = f["e070528/Neuron2/citronellal/stimOnset"][...][0]
train_list = [f[y][...] for y in
              ["e070528/Neuron2/citronellal/stim"+str(x)
               for x in range(1,16)]]
fig = plt.figure(figsize=(5,5))
raster_plot(train_list,citron_onset)
plt.xlim(-6,6)
plt.ylim(0,16)
plt.grid(True)
plt.xlabel("Time (s)",fontdict={'fontsize':18})
plt.ylabel("Trial",fontdict={'fontsize':18})
plt.savefig('figs/raster-plot-1.png')
plt.close()
'figs/raster-plot-1.png'
```



The second figure with raster plots is obtained with:

```
citron_onset = f["e060817/Neuron1/citronellal/stimOnset"][...][0]
fig = plt.figure(figsize=(5,10))
plt.subplot(211)
raster_plot(e060817citron[0],citron_onset)
plt.xlim(-2,6)
plt.ylim(0,21)
plt.grid(True)
plt.xlabel("Time (s)",fontdict={'fontsize':15})
plt.ylabel("Trial",fontdict={'fontsize':15})
plt.title("Citronellal",fontdict={'fontsize':20})
plt.subplot(212)
terpi_onset = f["e060817/Neuron1/terpineol/stimOnset"][...][0]
raster_plot(e060817terpi[0],terpi_onset)
plt.xlim(-2,6)
plt.ylim(0,21)
plt.grid(True)
plt.xlabel("Time (s)",fontdict={'fontsize':15})
plt.ylabel("Trial",fontdict={'fontsize':15})
plt.title("Terpineol",fontdict={'fontsize':20})
plt.subplots_adjust(wspace=0.4,hspace=0.4)
plt.savefig('figs/raster-plot-2.png')
plt.close()
'figs/raster-plot-2.png'
```

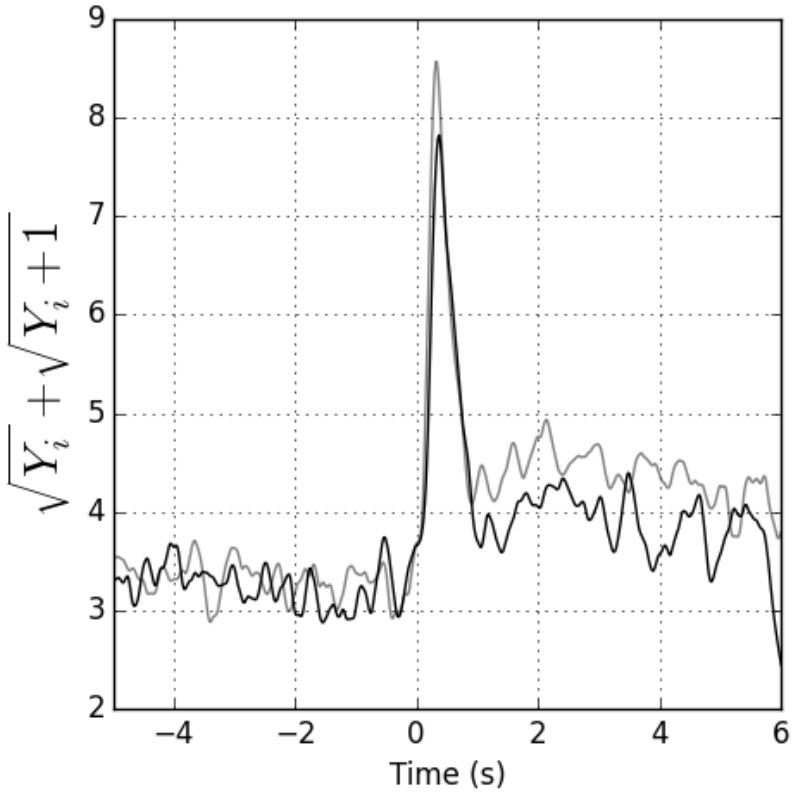

Citronellal
Trial
Time (s)
0
5
10
15
20
−2
−1
0
1
2
3
4
5
6


Terpineol
Trial
Time (s)
0
5
10
15
20
−2
−1
0
1
2
3
4
5
6

2.9 Terpineol and citronellal responses of neuron 1 from e060817

We create a figure showing the SmoothStabilizedPSTH instances:

```
fig = plt.figure(figsize=(5,5))
e060817terpi_sspsth[0].plot(what="smooth",color='grey')
e060817citron_sspsth[0].plot(what="smooth",color='black')
plt.xlim(-5,6)
plt.ylabel(r'$\sqrt{Y_i}+\sqrt{Y_i+1}$',fontdict={'fontsize':20})
plt.grid(True)
plt.savefig('figs/terpi-citron-SSPSTH-comp.png')
plt.close()
'figs/terpi-citron-SSPSTH-comp.png'
```



References

Cox, D. R. and P. A. W. Lewis (1966). *The Statistical Analysis of Series of Events*. John Wiley & Sons.

Durbin, J. (1961). "Some Methods of Constructing Exact Tests". In: *Biometrika* 48.1/2, pp. 41–55.

Kendall, W.S., J.M. Marin, and C.P. Robert (2007). "Brownian Confidence Bands on Monte Carlo Output". In: *Statistics and Computing* 17.1, pp. 1–10.

Lewis, Peter A. W. (1965). "Some Results on Tests for Poisson Processes". In: *Biometrika* 52.1/2, pp. 67–77.

Loader, C. R. and J. J. Deely (1987). "Computations of boundary crossing probabilities for the Wiener process". In: *Journal of Statistical Computation and Simulation* 27.2, pp. 95–105.

Marsaglia, George and John Marsaglia (2004). "Evaluating the Anderson-Darling Distribution". In: *Journal of Statistical Software* 9.2, pp. 1–5.

Pouzat, Christophe and Antoine Chaffiol (2009). "Automatic Spike Train Analysis and Report Generation. An Implementation with R, R2HTML and STAR". In: *J Neurosci Methods* 181, pp. 119–144.

— (2015). *Data set from Pouzat and Chaffiol (2009) Journal of Neuroscience Methods 181:119.* DOI:10.5281/zenodo.14281.

Pouzat, Christophe and Georgios Is. Detorakis (2014). "SPySort: Neuronal Spike Sorting with Python". In: *Proceedings of the 7th European Conference on Python in Science (EuroSciPy 2014)*. Ed. by Pierre de Buyl and Nelle Varoquaux, pp. 27–34.